TIBCO®
The Power of Now®

TIBCO Spotfire S+® 8.2 Programmer's Guide

November 2010

TIBCO Software Inc.

IMPORTANT INFORMATION

Reference

The correct bibliographic reference for this document is as follows:

TIBCO Spotfire S+® 8.2 Programmer's Guide, TIBCO Software Inc.

Technical Support

For technical support, please visit http://spotfire.tibco.com/support and register for a support account.

TIBCO SPOTFIRE S+ BOOKS

Note about Naming
Throughout the documentation, we have attempted to distinguish between the language (S-PLUS) and the product (Spotfire S+). • "S-PLUS" refers to the engine, the language, and its constituents (that is objects, functions, expressions, and so forth). • "Spotfire S+" refers to all and any parts of the product beyond the language, including the product user interfaces, libraries, and documentation, as well as general product and language behavior.

The TIBCO Spotfire S+® documentation includes books to address your focus and knowledge level. Review the following table to help you choose the Spotfire S+ book that meets your needs. These books are available in PDF format in the following locations:

- In your Spotfire S+ installation directory (**SHOME\help** on Windows, **SHOME/doc** on UNIX/Linux).
- In the Spotfire S+ Workbench, from the **Help ▶ Spotfire S+ Manuals** menu item.
- In Microsoft® Windows®, in the Spotfire S+ GUI, from the **Help ▶ Online Manuals** menu item.

Spotfire S+ documentation.

Information you need if you...	**See the...**
Must install or configure your current installation of Spotfire S+; review system requirements.	*Installtion and Administration Guide*
Want to review the third-party products included in Spotfire S+, along with their legal notices and licenses.	*Licenses*

Spotfire S+ documentation. (Continued)

Information you need if you...	**See the...**
Are new to the S language and the Spotfire S+ GUI, and you want an introduction to importing data, producing simple graphs, applying statistical models, and viewing data in Microsoft Excel®.	*Getting Started Guide*
Are a new Spotfire S+ user and need how to use Spotfire S+, primarily through the GUI.	*User's Guide*
Are familiar with the S language and Spotfire S+, and you want to use the Spotfire S+ plug-in, or customization, of the Eclipse Integrated Development Environment (IDE).	*Spotfire S+ Workbench User's Guide*
Have used the S language and Spotfire S+, and you want to know how to write, debug, and program functions from the **Commands** window.	*Programmer's Guide*
Are familiar with the S language and Spotfire S+, and you want to extend its functionality in your own application or within Spotfire S+.	*Application Developer's Guide*
Are familiar with the S language and Spotfire S+, and you are looking for information about creating or editing graphics, either from a **Commands** window or the Windows GUI, or using Spotfire S+ supported graphics devices.	*Guide to Graphics*
Are familiar with the S language and Spotfire S+, and you want to use the Big Data library to import and manipulate very large data sets.	*Big Data User's Guide*
Want to download or create Spotfire S+ packages for submission to the Comprehensive S-PLUS Archive Network (CSAN) site, and need to know the steps.	*Guide to Packages*

Spotfire S+ documentation. (Continued)

Information you need if you...	**See the...**
Are looking for categorized information about individual S-PLUS functions.	*Function Guide*
If you are familiar with the S language and Spotfire S+, and you need a reference for the range of statistical modelling and analysis techniques in Spotfire S+. Volume 1 includes information on specifying models in Spotfire S+, on probability, on estimation and inference, on regression and smoothing, and on analysis of variance.	*Guide to Statistics, Vol. 1*
If you are familiar with the S language and Spotfire S+, and you need a reference for the range of statistical modelling and analysis techniques in Spotfire S+. Volume 2 includes information on multivariate techniques, time series analysis, survival analysis, resampling techniques, and mathematical computing in Spotfire S+.	*Guide to Statistics, Vol. 2*

CONTENTS

THE S-PLUS LANGUAGE 1

INTRODUCTION TO SPOTFIRE S+

S-PLUS is a language specially created for exploratory data analysis and statistics. You can use Spotfire S+ productively and effectively without even writing a one-line program in the S-PLUS language. However, most users begin programming in Spotfire S+ 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 Spotfire S+ 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 Spotfire S+ 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 Spotfire S+, although a full discussion is deferred until Chapter 12, Object-Oriented Programming in Spotfire S+. We then describe the basic syntax and data types in S-PLUS. Programming in Spotfire S+ does not require, but greatly benefits from, programming tools such as text editors and source control. We touch on these tools briefly in the section The Spotfire S+ Programming Environment (page 14). Finally, we introduce the various graphics paradigms, and discuss when each should be used.

Interpreted vs. Compiled Languages

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

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 Spotfire S+ programs needs to have a compatible version of Spotfire S+.

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 Spotfire S+ 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 Spotfire S+ 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 Spotfire S+, *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 might define class-specific methods for the functions `draw`, `redraw`, and `move`. Spotfire S+ includes a mechanism for determining whether a function is generic, and if so, determining the class of its arguments and calling the appropriate method, so that, for example, if `draw` is generic and `orb` is an object of class `circle`, the call `draw(orb)` would automatically call the `draw` method for class `circle`, and draw `orb`.

We will take up object-oriented programming in detail in Chapter 12, Object-Oriented Programming in Spotfire S+.

Versions of the S Language

There are currently two distinct versions of the S language in common use: the S Version 3 language that underlies S-PLUS 2000 for Windows® (and all earlier versions of S-PLUS for Windows, as well as UNIX® versions of S-PLUS from 3.0 to 3.4) and the S Version 4 language that underlies S-PLUS 5.0 and later on UNIX and S-PLUS 6 for Windows and later.

The S Version 3 language (referred to in this document as SV3) introduced the modeling language that is the foundation for most Spotfire S+ statistical and analytic functionality. It had a simple object-oriented structure with a dispatch mechanism built on naming conventions. It did not apply any class structure to existing S-PLUS objects such as vectors and matrices.

The S Version 4 language (referred to in this document as SV4) introduced a strongly-typed object-oriented structure similar to that in C++; in SV4, all objects are assigned classes, the dispatch mechanism for methods is much more sophisticated, and there are far stricter controls over inheritance. In particular, multiple inheritance is no longer supported. If you are new to Spotfire S+, you will be using SV4 from the start and you will find the instructions in this manual accurate. If you have been working with Spotfire S+ a while, you may find that certain S expressions you may have used in the past yield different answers under SV4. We have tried to cover most of the serious differences in output between SV3 and SV4 in the appendix on migrating your code to the SV4 version of S-PLUS (6.0 and higher) in the *User's Guide.*

Programming Tools in Spotfire S+

In the Microsoft Windows® GUI, there are two main tools for developing Spotfire S+ 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, found on the **Standard** toolbar.*

The > prompt in the **Commands** window indicates Spotfire S+ is ready for your input. You can now type expressions for Spotfire S+ to interpret. Throughout this book, we show typed commands preceded by the Spotfire S+ 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 Spotfire S+ altogether.

The `fix` function is available from the **Commands** window to let you edit a function within a text editor.

In the Microsoft Windows GUI **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 Spotfire S+ programs, and for building programs from a variety of sources, such as the History log.

For your first sessions programming in Spotfire S+, we recommend you use the **Commands** window.

SYNTAX OF S-PLUS EXPRESSIONS

You interact with Spotfire S+ by typing *expressions*, which the Spotfire S+ interpreter evaluates and executes. Spotfire S+ 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)
.Call("S_c_use_method", "sqrt")
```

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 Spotfire S+ interpreter as names. Spotfire S+ 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 2, Data Management.

Spotfire S+ 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:

```
> plot(corn.rain)
> mean(corn.rain)
[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` (in Microsoft Windows®) or `motif` (in UNIX), `plot`, and `q` are called primarily for their side effects, such as starting or closing a graphics device, plotting points, or ending a Spotfire S+ 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), Spotfire S+ 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 equivalents, the equal sign, =, and 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 Spotfire S+ 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`.[1] Assignments typed at the Spotfire S+ prompt are *permanent*; objects created in this way endure from session to session, until removed.

1. The complete list is as follows: `if`, `is`, `else`, `for`, `while`, `repeat`, `next`, `break`, `in`, `function`, `return`, `TRUE`, `T`, `FALSE`, `F`, `NULL`, `NA`, `Inf`, `NaN`.

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 2, Data Management.

Object names must begin with a letter or period, 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. With S-PLUS 5.x and later, another naming convention arose, where words previously separated by periods were simply concatenated, with the second and subsequent words having initial caps, as in the following: `exportData`, `getCurrDirectory`, `findClassObjects`.

Subscripting

Another common operator is the *subscript* operator [, used to extract subsets of a 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 T F F 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 Spotfire S+ because it emphasizes treating data objects as whole entities, rather than as collections of individual observations. This point of view is central to Spotfire S+ utility as a data analysis computing environment. For a full discussion of subscripting, see the Operating on Subsets of Data on page 172.

DATA CLASSES

Everything in S-PLUS is an *object*. All objects have a *class*. A S-PLUS expression (itself an object) is interpreted by the Spotfire S+ evaluator and returns a value, another object that can be assigned a name. An object's class determines the *representation* of the object, that is, what types of information can be found within the object, and where that information can be found. Most information about an object is contained within specialized structures called *slots*.

The simplest data objects are one-dimensional arrays called *vectors*, consisting of any number of *elements* corresponding to individual data points. The simplest elements are literal expressions that, either singly or matched like-with-like, produce the following classes:

- `logical`: The values `T` (or `TRUE`) and `F` (or `FALSE`).
- `integer`: Integer values such as `3` or `-4`.
- `numeric`: Floating-point real numbers (double-precision by default). Numerical values can be written as whole numbers (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 integers or numeric (for example, `3 + 1.23i`).
- `character`: character strings enclosed by matching double quotes (") or apostrophes ('), for example, `"Alabama"`, `'idea'`.

These simple element classes are listed in order from least informative (`logical`) to most informative (`character`); this order is important when considering data objects formed by combining elements from different classes.

The number of elements in a data object determines the object's *length*. A vector of length 1 can be created simply by typing a literal and pressing ENTER:

```
> 7.4
[1] 7.4
> "hello"
[1] "hello"
```

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 classes into a single vector, Spotfire S+ coerces all the elements to the most informative class[1]:

```
> c(T, 8.3, 5)
[1] 1.0 8.3 5.0
> c(8.3, 9 + 6i)
[1] 8.3+0i 9.0+6i
> c(T, 8.3, "hello")
[1] "TRUE" "8.3" "hello"
```

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

```
> class(c(T, 8.3, 5.0))
[1] "numeric"
> length(c(T, 8.3, "hello"))
[1] 3
```

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 class `"list"`) using the `list` function:

1. This statement about coercion applies strictly only to the five simple classes described on page 11. These simple classes correspond roughly to what S version 3 and earlier referred to as *modes.* (Although objects of class `"integer"` have mode `"numeric"`.) The concept of modes persists in S version 4, but it has been almost entirely superseded by the new class mechanism.

```
> 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` class is an extremely powerful tool in Spotfire S+, and we shall use it extensively throughout this book.

THE SPOTFIRE S+ PROGRAMMING ENVIRONMENT

Spotfire S+ uses tools available in the Microsoft Windows® environment. Some of these tools are built into Spotfire S+ as functions–for example, the `edit` function (Windows) and `vi` function (UNIX®), which allows you to edit with the Windows Notepad in Windows and `vi` editor in UNIX, respectively. Windows software, including spreadsheets such as Microsoft Excel and word processors such as Microsoft Word, can be called from Spotfire S+ using the `dos` and `system` functions. UNIX system functions, such as `ls` and `awk`, can be called via the `unix` function.

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 Spotfire S+ data by using the `fix` function.

```
> fix(x)
```

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

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

where *editor* is the binary executable (.exe in Windows) or command that runs your favorite text editor. To set this option for each Spotfire S+ session, add the expression to your `.First` function. In Spotfire S+, this option defaults to Windows Notepad or, in UNIX, `options()$editor`.

Once you've set up Spotfire S+ 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 Spotfire S+. 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 Data Output on page 208) 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 a Spotfire S+ *script*, which is just a text file containing valid S-PLUS expressions.

You can run Spotfire S+ scripts in any of the following ways:

1. The `source` function in the **Commands** window.
2. On Windows, loading it into a **Script** window, highlighting the required code and clicking the **Run** button on the Script toolbar. See Chapter 9, Using the Script and Report Windows, in the *User's Guide*.
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

Spotfire S+ 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 a Spotfire S+ script, which you can read or edit just like any text file. Objects dumped by `data.dump` result in files that are *not* Spotfire S+ 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 Spotfire S+ transport file format (see Chapter 9, Importing and Exporting, for more details).

GRAPHICS PARADIGMS

In Spotfire S+, there are three basic graphics paradigms for Windows which we refer to as Editable Graphics, Traditional Graphics, and Traditional Trellis Graphics. UNIX only has two and does not include the capability cited below in the section (Windows) Converting Non-editable Graphics to Editable Graphics on page 18),

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.

Chapter 4, Editable Graphics Commands, in the *Guide to Graphics*, describes these graphics.

Traditional Graphics

Traditional S-PLUS language functions are available to create plots identical to those in previous versions of Spotfire S+.

Chapter 2, Traditional Graphics in the *Guide to Graphics* , describes these graphics.

Traditional Trellis Graphics

The Trellis graphics paradigm provides multipanel conditioning to effectively discover relationships present in data. These graphics were implemented using calls to the traditional graphics language functions.

Chapter 3, Traditional Trellis Graphics in the *Guide to Graphics*, describes using conditioning with the object-oriented plots.

(Windows) Converting Non-editable Graphics to Editable Graphics

By default, traditional graphics commands 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 Graphs button on the **Commands** window toolbar. Alternately, you may open a Graph sheet device in editable graphics mode using

```
> graphsheet(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 Spotfire S+ 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-date 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 Spotfire S+. 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 is a powerful technique for exploring multivariate structure in data. They were implemented in traditional graphics for convenience and to make them available to all Spotfire S+ users. This implementation is described in Chapter 3, Traditional Trellis graphics in the *Guide to Graphics.*

Trellis conditioning has been incorporated directly into the editable graphics system graphics menu, making the power of multipanel conditioning available in all editable GUI-created 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.

These graphics, introduced in S-PLUS 4 for Windows, are not available to users running Spotfire S+ for UNIX. If you intend to make your functions available to users on both Windows and UNIX platforms, you need to use traditional graphics and traditional Trellis graphics rather than editable graphics.

DATA MANAGEMENT 2

INTRODUCTION

What happens when you associate a S-PLUS object with a name? How does Spotfire S+ find the object again once assigned? For the most part, when you are using Spotfire S+, 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 Spotfire S+ maintains data objects can be helpful in debugging. Knowledge of Spotfire S+ data management can also help you write more efficient functions, a topic we will turn to in Chapter 14, Using Less Time and Memory.

This chapter discusses the main features of Spotfire S+ 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 an expression is evaluated in a certain frame, we are saying that the evaluation uses that frame's associated names and values.

Frames are important in Spotfire S+ because they serve to limit the scope of assignments.

Note

S Programming (Venables & Ripley, 2000) defines S scoping rules for resolving S expressions as follows:

- Objects not found in the current (*local*) frame are next referred to the frame of the t*op-level* expression, then the frame of the *session*, and then on the *search path*. Note that objects created in the parent functions are not found in the search.

Assignments associate names with values, and the Spotfire S+ evaluator uses this association to determine the values associated with each name during the evaluation of a function. New frames are created by 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.

Note

Scoping rules vary between the R and S languages. For more information about differences between R and S, see the *Guide to Spotfire S+ Packages*, or see the discussion on the Web site **http://spotfire.tibco.com/csan**.

Every time you type an expression on the Spotfire S+ command line, Spotfire S+ creates a frame called the *task frame, expression frame*, *top-level frame*, or simply *frame 1*. The task frame initially contains the unevaluated expression and a flag specifying whether automatic print is enabled or disabled. If the expression is a function call, Spotfire S+ 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)`, then just before `fcn.B` completes, its frame looks like the following when we use `sys.frame` to display the frame:

```
$a:
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068
 [6] 2.449490 2.645751 2.828427 3.000000 3.162278

$b:
 [1] 2.302585 2.484907 2.639057 2.772589 2.890372
 [6] 2.995732 3.091042 3.178054 3.258097 3.332205

$x:
 [1]  1  2  3  4  5  6  7  8  9 10

$y:
 [1] 10 12 14 16 18 20 22 24 26 28

$C:
 [1] 3.302585 3.899120 4.371108 4.772589 5.126440
 [6] 5.445222 5.736794 6.006481 6.258097 6.494482
```

```
$.Auto.print:
[1] F
```

Expressions that do not involve assignment (such as `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 task 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 2.1 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)
}
```

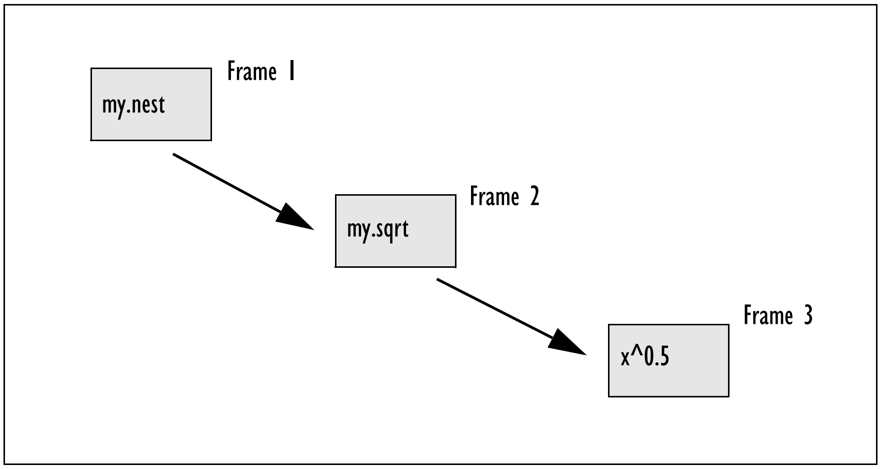


Figure 2.1: *The frame hierarchy.*

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


[[2]]:
[[2]]$expression:
expression(my.nest(4))

[[2]]$local:
[1] T

[[2]]$parent:
NULL


[[3]]:
[[3]]$x:
[1] 4
```

```
[[4]]:
[[4]]$x:
[1] 4
```

Here we see the expression frame in component `[[2]]`: it contains the auto-print 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 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] 3
```

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.lev.func <-
function(x)
{
  a <- sqrt(x)
  next.lev.func()
}

next.lev.func <-
function()
{
  get("a", frame = sys.parent()) * 2
}
```

When we call top.lev.func, we obtain the following result:

```
> top.lev.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 28). Each function is evaluated in its own frame.

Frames and Argument Evaluation

Default values for arguments are evaluated in the function's frame, while values for named arguments are evaluated in the *parent* frame. When a function is called, arguments are placed, unevaluated, into the function's evaluation frame. As soon as the calling function references the argument, Spotfire S+ evaluates the named 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, Spotfire S+ 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 Spotfire S+ *lazy evaluation*, described in Chapter 7, Writing Functions in Spotfire S+. Default values, however, are evaluated in plot's frame.

Creating and Moving Frames

Most frames in Spotfire S+ 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. Data frames are lists, so `local` is often a data frame. Thus, for example, suppose we have a simple list object `myframe`:

```
> myframe <- list(a=100, b=30)
```

In evaluating the following, Spotfire S+ uses `myframe` as the frame for evaluation:

```
> eval(expression(a + b), local=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:

```
manycalc <-
function()
{ n <- new.frame(myframe)
  a <- eval(expression(max(a, b)), n)
  b <- eval(expression(mean(c(a, b))), n)
  clear.frame(n)
  a - b
}
```

The output from a call to `manycalc` is shown below:

```
> manycalc()
[1] 35
```

In this example, we take advantage of `myframe` to reuse the variable names `a` and `b` in `manycalc`'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 `manycalc`'s frame, yields the answer, 35. We used the `clear.frame` function to get rid of frame `n` when we were done with it. The frame is cleared automatically at the end of the function that creates it unless `move.frame` is called.

DATABASES IN SPOTFIRE S+

A Spotfire S+ *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 Spotfire S+ session. Three types of databases are in common use: ordinary chapters, which are directories that contain databases; meta directories, which are directories used in S-PLUS 5.x and later to hold S-PLUS objects that store class information, generic function definitions, and method definitions; and recursive (list-like) objects, particularly data frames. Other database types exist, but these three types are adequate for most purposes. If an object is referred to in a S-PLUS expression, and its name-value binding is not found in the current frame, Spotfire S+ searches the expression frame. If the binding is not found there, Spotfire S+ searches through databases, starting with database 0 and continuing along a user-specified search path. When you start Spotfire S+, this path consists of your *working directory* and several directories of S-PLUS functions and data sets. You can see the search path at any time using the `search` function:

```
> search()
 [1] "/homes/username/MySwork"
 [2] "splus"
 [3] "stat"
 [4] "data"
 [5] "trellis"
 [6] "nlme3"
 [7] "menu"
 [8] "sgui"
 [9] "winspj"
[10] "main"
```

Databases can be added to the search list with the `attach` function:

```
#in Windows:
> attach("c:/myfiles/.Data")
> search()
 [1] "C:\\DOCUME~1\\MYDOCU~1\\Spotfi~1\\Project1"
```

```
  [2] ".Data"
  [3] "splus"
  [4] "stat"
  [5] "data"
  [6] "trellis"
  [7] "nlme3"
  [8] "menu"
  [9] "sgui"
 [10] "winspj"
 [11] "main"
```

Similarly, databases can be removed from the search list with the `detach` function:

```
# in Windows:
> detach("c:/myfiles/.Data")
> search()
  [1] "C:\\DOCUME~1\\MYDOCU~1\\Spotfi~1\\Project1"
  [2] "splus"
  [3] "stat"
  [4] "data"
  [5] "trellis"
  [6] "nlme3"
  [7] "menu"
  [8] "sgui"
  [9] "winspj"
 [10] "main"
```

Databases are generally referred to either by number (that is, their position in the search list) or by name (again, the name as returned by `search`). A third way is to use an object of class `"attached"`, which are the objects returned by `attach`, `library`, and `module`. You can save these return objects in your database; they are valid for the full time the database is attached. If you don't assign these objects when you attach the database, they can be recalled later by calling `database.attached` with the position or name of the database as its only argument.

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:

```
# in Windows:
> attach("c:/myfiles/.Data", pos=8)
> search()
 [1] "C:\\DOCUME~1\\MYDOCU~1\\Spotfi~1\\Project1"
 [2] "splus"
 [3] "stat"
 [4] "data"
 [5] "trellis"
 [6] "nlme3"
 [7] "menu"
 [8] "c:/myfiles/.Data"
 [9] "sgui"
[10] "winspj"
[11] "main"
```

You can also provide an "alias" for a directory with the `name` argument:

```
# in Windows:
> attach("c:/myfiles/funcs", pos=2, name="myfuncs")
> search()
 [1] "C:\\DOCUME~1\\MYDOCU~1\\Spotfi~1\\Project1"
 [2] "myfuncs"
 [3] "splus"
 [4] "stat"
 [5] "data"
 [6] "trellis"
 [7] "nlme3"
 [8] "menu"
 [9] "sgui"
[10] "winspj"
[11] "main"
```

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("\\myfiles\\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 2.1: *Functions to manipulate the contents of databases.*

Function	Purpose
`exists`	Tests whether a given object exists in the search path.
`get`	Returns a copy of the object, if it exists. Otherwise, returns an error.
`assign`	Creates a new name-value pair in a specified database.
`remove`	Deletes specified objects, if they exist, from the specified database.
`objects`	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 7, Writing Functions in Spotfire S+, 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 Spotfire S+ 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
Problem: Syntax error: illegal name ("do") on input line 1
```

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, Spotfire S+ 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.

Meta Databases

Meta databases store S-PLUS objects that describe the classes and methods available in S-PLUS. Calls to functions such as setMethod or setClass perform assignment in these meta databases. Every Spotfire S+ chapter includes, beneath its .Data directory, a __Meta directory. (Notice that the name is prefaced by two underscores.) You can view the contents of the meta directory by using the objects function with both the where and meta=1 arguments:

```
> objects("splus", meta=1)
 [1] ".Generics"              "C#DTDRef"
 [3] "C#clipboardConnection" "C#fdConnection"
 [5] "C#fillText"             "C#groupVec"
 [7] "C#groupVecVirtual"      "C#numericSequence"
 [9] "C#parseTest"            "C#positions"
[11] "C#positionsCalendar"    "C#positionsNumeric"
[13] "C#regularExpression"    "C#series"
[15] "C#seriesVirtual"        "C#sgml"
[17] "C#sgmlEmptyTag"         "C#sgmlMap"
...
```

The objects with names beginning with `C#` contain class definition information. Method definitions are in objects with names beginning with `M#`.

Meta databases can be explored and manipulated with the standard database tools (`get`, `exists`, etc.), provided the `meta=1` argument is supplied. If you need to modify objects in the `__Meta` directory, use functions such as `getClass`, `setClass`, `getMethod`, `setMethod`, etc.

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 Spotfire S+ session are not reflected in the dictionary. You can look at the dictionary for any database with the `objects` function:

```
> objects("C:\\Documents and Settings\\My
Documents\\Spotfire S+ Projects\\Project1")
 [1] "myframe"     "manycalc"    "x"           "2do"
 [5] "assn.use"    "_1"          "airtemp.jit" "air.jit"
 [9] "f"           "illit"       "murder"      "circle"
> dbobjects(kyphosis)
[1] "Number"   "Kyphosis" "Start"    "Age"
```

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 Spotfire S+. 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 Spotfire S+. 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 Spotfire S+. Spotfire S+ 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, Spotfire S+ 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, Spotfire S+ 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, Spotfire S+ 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] "__META    0    01-21-93 3:29p"
```

```
[9] "___NONFI   42  01-21-93 5:24p"
[10] "__1 156 01-28-93 2:02p"
[11] "   7 file(s)     32345 bytes"
[12] "              131129344 bytes free"
> "__1" <- 10:20
> dos("DIR .data")
. . .
[9] "___NONFI 42 01-21-93 5:24p"
[10] "__2 234 01-28-93 2:05p"
[11] "__1 156 01-28-93 2:02p"
. . .
```

Here "either/or" needs to be mapped because DOS does not permit file names with slashes, so Spotfire S+ maps the object to the file __1 and records the true file name and the mapped object name in the file ___nonfile. The new object "__1", while having a perfectly valid file name, conflicts with the Spotfire S+ 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")
. . .
[9] "___NONFI 42 01-21-93 5:24p"
[10] "__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 ___nonfile 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 23). 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 `kyphosis` data frame, you could use the following:

```
> kyphosis$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 `kyphosis` as a database, however, you can access its components as whole objects:

```
> attach(kyphosis)
> Age
  1   2   3 4 5 6  7  8   9 10 11  12 13 14  16 17 18  19 20
 71 158 128 2 1 1 61 37 113 59 82 148 18  1 168  1 78 175 80
  . . .
```

The simplified syntax is useful when you are constructing complicated expressions involving different components. For example, compare the following expressions:

```
if (any(kyphosis$Age > 130))
  sqrt(kyphosis$Start)
else exp(kyphosis$Number)

if (any(Age > 130))
  sqrt(Start)
else exp(Number)
```

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 Spotfire S+ session with a data frame attached in position 1.

MATCHING NAMES AND VALUES

When Spotfire S+ 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, Spotfire S+ searches in turn each of the meta databases currently in the search path, starting with the working directory, then the regular databases. 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:

```
fcn.C <-
function()
{
  x <- 3
  fcn.D()
}
fcn.D <-
function()
{
  x^2
}
```

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. 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 to the working data alter the contents of those data, 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()
Problem in fcn.E(): Any error
Use traceback() to see the call stack
> 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 = 1)
  print(2 * get("x", where = 1))
  stop("Nothing is committed")
}
```

The value printed by the second line of `assn.use` reflects the assignment to `x` on database 1, even though `x` has not yet been permanently committed:

```
> assn.use()
[1] 20 18 16 14 12 10 8 6 4 2
Problem in assn.use(): Nothing is committed
Use traceback() to see the call stack
```

Because assignments remain pending until the completion of the top-level expression, Spotfire S+ 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 Spotfire S+. (Similar memory growth can occur simply from *reading* too many data objects, because by default Spotfire S+ stores any newly-read object in memory in case it is needed elsewhere in the top-level expression.)

Note

The elimination of backout protection for databases other than database 1 is a significant change for S-PLUS 5.x and later. As you migrate your functions forward, be careful that you are not relying on such protection.

Another way to override the backout mechanism is to use the `synchronize` function, with no arguments. When called with no arguments, `synchronize` simply commits all pending assignments to the working data. If called with a numeric vector *i* as its argument, `synchronize` tells Spotfire S+ to reattach the databases given by *i* in the search path. Reattaching not only updates the dictionaries and databases with respect to commitment, but also with respect to changes (if any) made by other processes. Note, however, that the reattachment takes place only at the end of the top-level expression. Thus, you cannot (as in S-PLUS 4.x and earlier) use `synchronize` to synchronize specific databases in the middle of a function call.

COMPUTING ON THE LANGUAGE

3

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, Spotfire S+ 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 Spotfire S+ calculation results in a new S-PLUS object, and objects can always be manipulated in Spotfire S+. Chapter 2, Data Management, discusses several uses of these techniques.

Computing on the language is useful for a number of tasks, including the following:

- Symbolic computations.
- Making labels and titles for a graphics plot, using part or all of the expression used to create the plot.
- Creating file names and object names.
- 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 described in Chapter 11, Debugging Your Functions).

Most of these tasks involve some variant of the following basic technique:

1. Create an unevaluated expression, with `substitute`, `expression`, `parse`, or `Quote` (which is like `expression`).
2. Perform some intermediate computations.

3. Generate finished output, using either or both of the following methods:
    - Deparsing the unevaluated expression (with `deparse`).
    - 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:

$$\frac{d}{dx}\sin(x) = \cos(x)$$

$$\int(x^2 + 3x + 4)dx = \frac{x^3}{3} + \frac{3x^2}{2} + 4x + C$$

To perform symbolic computations in Spotfire S+, you must interrupt the usual Spotfire S+ pattern of evaluation to capture *unevaluated* expressions to represent formulas and then perform the desired manipulations and return the result as a 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`. Spotfire S+ returns the actual argument corresponding to that formal name in the current function call.

For example, S-PLUS has a function, `D`, that takes a 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. For example, 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. Spotfire S+ 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 *deparses* 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, short=T)
  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 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 Spotfire S+ 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 of the `fix` function, which is 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 Spotfire S+ 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 Spotfire S+.

The following function makes it easy to move functions (or other objects) between directories. Here the default destination for the moved object is a Spotfire S+ chapter labeled **.Sutilities**:

```
move <-
function(object, from = 1, to = ".Sutilities")
{
  objname <- deparse(substitute(object))
  assign(objname, get(objname, where = from),
     where = to)
  remove(objname, where = from)
}
```

BUILDING EXPRESSIONS AND FUNCTION CALLS

Like lists, expressions, function definitions, and function calls are all recursive S-PLUS objects 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 and then use `eval` to evaluate the expression or call. The following subsections give an introduction to this technique.

Building Unevaluated Expressions

As we have seen, the `substitute` function 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.rain)
> fexpr
expression(3 * corn.rain)
> 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.choice <- expression(Gaussian = rnorm(1),
      Uniform = runif(1), Exponential = rexp( 1),
      Cauchy = rcauchy(1))
  pick <- menu(names(rand.choice))
  if(pick)
      eval(rand.choice[pick])
}
```

The `expression` function returns an object of class `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 the 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" "include.lowest" "..."    "xlab"
[9] ""
```

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" "include.lowest" "..."    "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 11, 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) and then look at the definition of `argnames`, we see the call to `browser` embedded:

```
> trace(argnames,browser)
> argnames
function(funcname)
{
  if(.Traceon) {
      assign(".Traceon", F, frame = 0)
      on.exit(assign(".Traceon", T, frame = 0))
      cat("On entry: ")
      browser()
      assign(".Traceon", T, frame = 0)
  }
  {
      names(funcname)[ - length(funcname)]
  }
}
```

Here is a simplified version of `trace`, called `simp.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)
{
        assign(".Traceon", F, frame = 0)
        on.exit(assign(".Traceon", T, frame = 0))
        cat("On entry: ")
        browser()
        assign(".Traceon", T, frame = 0)
   }
[[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 `simp.trace` function is shown below:

```
simp.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){
        assign(".Traceon", F, frame = 0)
        on.exit(assign(".Traceon", T, frame = 0))
        cat("On entry: ")
        browser()
        assign(".Traceon", T, frame = 0)
    }
    )[[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.rain))[[1]]
hist(corn.rain)
```

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 and then accepts arbitrary arguments as arguments to the function:

```
> call("hist", corn.rain)
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 Spotfire S+ from reading the arguments into memory until evaluation, the idiom `as.name("`*`argument`*`")` can be useful:

```
> call("hist", as.name("corn.rain"))
hist(corn.rain)
```

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 = "yule-walker")
{
  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" ),
     "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 Spotfire S+ programming, however, is to create an ordinary *list* of the appropriate form and then change the *mode* of the list to `call`. This idiom, in fact, is used by the actual `ar` function distributed with Spotfire S+:

```
> 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 class 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` and 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 = "yule-walker")
{
  arglist <- list(x = x, aic = as.logical(aic))
  if(!missing(order.max))
     arglist$order.max <- order.max
  imeth <- charmatch(method, c("yule-walker", "burg"),
     nomatch = 0)
  method.name <- switch(imeth + 1,
     stop("method should be either yule-walker or burg" ),
     as.name("ar.yw"),
     as.name("ar.burg"))
  ar <- eval(method.name,
     local = sys.parent(1))( unlist(arglist))
  ar$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 2, 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 and 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 and 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 (that is, 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, use the function `match.call`.

As we saw in the previous section, the `match.call` function 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 ... formal argument, an optional argument to `match.call` can be used to specify whether the ... 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"))
```

The `match.call` function returns *unevaluated* arguments and is easy to use for routine manipulations. 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())
  . . .
}
```

DATA OBJECTS 4

INTRODUCTION

When using Spotfire S+, you should think of your data sets as *data objects* belonging to a certain *class*. Each class has a particular *representation*, often defined as a named list of *slots*. Each slot, in turn, contains an object of some other class.

To inspect an object, you can use one of two functions:

- `getSlots`
- `slotNames`

These functions return information about the slots. For example:

```
> slotNames(djia)

 [1] "data"              "positions"          "start.position"
 [4] "end.position"       "future.positions"  "units"
 [7] "title"             "documentation"      "attributes"
[10] "fiscal.year.start" "type"
```

The class of an object defines how the object is represented and determines what actions may be performed on the object and how those actions are performed. Among the most common classes of data objects are `numeric`, `character`, `factor`, `list`, and `data.frame`.

The simplest type of data object in Spotfire S+ is the *atomic vector*, a one-way array of *n* elements of a single *mode* (for example, numbers) that can be indexed numerically. Atomic vectors are so called to indicate that in Spotfire S+ they are indeed fundamental objects. All of Spotfire S+'s basic mathematical operations and data manipulation functions are designed to work on the vector *as a whole*, although individual elements of the vector can be extracted using their numerical indices.

More complicated data objects can be constructed from atomic vectors in one of two basic ways:

1. By allowing complete S objects as elements, or
2. By building new data classes from old using *slots*

Objects that contain other S objects as elements are called *recursive objects* and include such common S-PLUS objects as lists and data frames. A *list* is a vector for which each element is a distinct S object, of any type. A *data frame* is essentially a list in which each of the elements is an atomic vector, and all of the elements have the same length. With slots, you can uniquely define a new class of data object by storing the defining information (that is, the object's *attributes*) in one or more slots.

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 Spotfire S+.

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 the parts of a vector. To extract individual elements, use their numerical indices with the subscript operator `[`:

```
> car.gals[c(1,3,5)]
[1] 13.3 11.5 14.3
```

All elements within an atomic vector must be from only one of seven atomic *modes*—`logical`, `numeric`, `single`, `integer`, `complex`, `raw`, or `character`. (An eighth atomic mode, `NULL`, applies only to the `NULL` vector.) The number of elements and their mode completely define the data object as a vector. The class of any vector is the mode of its elements:

```
> class(c(T,T,F,T))
[1] "logical"
> class(c(1,2,3,4))
[1] "integer"
> class(c(1.24,3.45, pi))
[1] "numeric"
```

The number of elements in a vector is called the `length` of the vector and can be obtained for any vector using the `length` function:

```
> length(1:10)
[1] 10
```

Coercion of Values

When values of different modes are combined into a single atomic object, Spotfire S+ converts, or *coerces*, all values to a single mode in a way that preserves as much information as possible. The basic modes can be arranged in order of increasing information—`logical`, `integer`, `numeric`, `complex`, and `character`. 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"
```

Spotfire S+ 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 integers, `TRUE` values become the integer 1 and `FALSE` values become the integer 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 `integer` or `numeric` computations.

Creating Vectors

If you want to create a vector, 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.

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

Table 4.1: *Useful functions for creating vectors.*

Function	Description	Examples
`scan`	Reads values, any mode	`scan(), scan("data")`
`c`	Combines values, any mode	`c(1,3,2,6), c("yes","no")`
`rep`	Repeats values, any mode	`rep(NA,5), rep(c(1,2),3)`
`:`	`numeric` sequences	`1:5, 1:-1`
`seq`	`numeric` sequences	`seq(-pi,pi,.5)`
`vector`	Initializes vectors	`vector('complex',5)`
`logical`	Initializes `logical` vectors	`logical(3)`
`integer`	Initializes `integer` vectors	`integer(4)`
`numeric`	Initializes `numeric` vectors	`numeric(5)`
`complex`	Initializes `complex` vectors	`complex(6)`
`character`	Initializes `character` vectors	`character(7)`

Naming Vector Elements

You can assign names to vector elements 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, Spotfire S+ gives an error message.

STRUCTURES

Next in complexity after the atomic vectors are the *structures*, which, as the name implies, extend vectors by imposing a structure, typically a multi-dimensional array, upon the data.

The simplest structure is the two-dimensional *matrix*. A matrix starts with a vector and then adds the information about how many rows and columns the matrix contains. This information, the *dimension*, or `dim`, of the matrix, is stored in a slot in the representation of the `matrix` class. All structure classes have at least one slot, `.Data`, which must contain a vector. The classes `matrix` and `array` have one additional required slot, `.Dim`, to hold the dimension and one optional slot, `.Dimnames`, to hold the names for the rows and columns of a matrix and their analogues for higher dimensional arrays. Like simple vectors, structure objects are atomic, that is, all of their values must be of a single mode.

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` slot, 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` slot, as we see in the examples below.

Creating Matrices

To create a matrix from an existing vector, use the `dim` function to set the `.Dim` slot. 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`. 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, Spotfire S+ 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 of `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.

Naming Rows and Columns

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 lowercase and uppercase, 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:

```
> dimnames(mat) <- list(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 `dimnames` as follows:

```
> mat2 <- matrix(1:12, ncol=4,
+ dimnames=list(NULL,paste("col",1:4)))
```

Arrays

Arrays generalize matrices by extending the `.Dim` slot 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 Spotfire S+. 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` slot 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` slot 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 Spotfire S+, use the `array` function. The `array` function is analogous to `matrix`. It takes data and the appropriate dimensions as arguments to produce the array. If no data are 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` slot 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` slot 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 Naming Rows and Columns on page 73.

LISTS

A list is a completely flexible means for representing data. In earlier versions of S, it was the standard means of combining arbitrary objects into a single data object. Much the same effect can be created, however, using the notion of slots.

Up to this point, all the data objects 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 slots of an array may contain both the dimension (a numeric vector) and the `.Dimnames` slot (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" "Versicolor" "Virginica"
```

The value returned by `attributes` is a simple example of an S-PLUS *list*. Lists are a very general data type. They 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, the value of the `.Dim` slot, is a numeric vector of length three. The `dimnames` component, the value of the `.Dimnames` slot, 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 dollar sign (`$`).

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.

Naming Components

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",
+ "descrip")
> names(heart.list)
[1] "group" "total heart weight" "descrip"
```

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`, and `divorced`.
- *Experimental status*, where the values might be `treatment` and `control`.

Categorical data in Spotfire S+ is represented with a data type called a `factor`. The built-in data frame `fuel.frame` has a variable named `Type` that classifies each automobile as one of `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 that 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 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 3 3
[51] 2 2 2 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 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 3 3
[51] 2 2 2 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 that 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:

```
> classlist <- c("male", "female", "male", "male", "male",
+ "female", "female", "male", "female", "male")
```

```
> factor(classlist)
[1] male female male male male female female male
[9] female male
```

Spotfire S+ creates two levels with labels `female` and `male`, respectively.

Table 4.2: *Arguments to* `factor`.

Argument	Description
`x`	Data, to be thought of as taking values on the finite set of levels.
`levels`	Optional vector of `levels` for the factor. The default value of `levels` is the *sorted* list of distinct values of `x`.
`labels`	Optional vector of values to use as labels for the `levels` of the factor. The default is `as.character(levels)`.
`excludes`	A vector of values to be excluded from forming `levels`.

The `levels` 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 `levels` argument. Any values omitted from the `levels` argument are considered missing.

```
> intensity <- factor(c("Hi","Med","Lo","Hi","Hi","Lo"),
+ levels = c("Lo","Hi"))
> intensity
[1] Hi NA Lo Hi Hi Lo
> levels(intensity)
[1] "Lo" "Hi"
```

If you had left the `levels` argument off, the levels would have been ordered alphabetically as `Hi`, `Low`, `Medium`. You use the `labels` argument if you want the levels to be something other than the original data.

```
> factor(c("Hi","Lo","Med","Hi","Hi","Lo"),
+ levels=c("Lo","Hi"), labels = c("LowDose","HighDose"))
[1] HighDose LowDose NA HighDose HighDose LowDose
```

Warning

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

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

```
> factor(c("Hi","Med","Lo","Hi","Hi","Lo"),
+ exclude =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"),
+ levels=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 `levels` argument determines the order placed on the levels.

Warning

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

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 breakpoints or an integer specifying how many groups to divide the numerical data into; `cut` then 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 `High` 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 `labels` argument.

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

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
```

TIME SERIES AND SIGNAL BASICS

5

INTRODUCTION

Time series and signal data sets have single or multivariate data columns that are associated with a time-, space-, or frequency-domain set of ordered positions, where the positions are an important feature of the values and their analysis. This type of data can arise in many contexts. For example, in the financial marketplace, trading tickers record the price and quantity of each trade, and each takes place at a particular time; these data can be analyzed for use in making market predictions. Weekly or monthly measurements of sunspot activity can be used to study cycles in sunspot activity. Electrical real-time digital signal data (or its Fourier transform, frequency-ordered data) can be used to study the properties of waveguides.

This chapter describes how to create, manipulate, and plot time series and signals in Spotfire S+. Chapter 27 of the *Guide to Statistics, Volume 2*, Analyzing Time Series and Signals, covers time series and signal analysis, and Chapter 6, Dates, Times, Time Intervals, and Sequences, contains more information specific to manipulating times and dates and creating time and numeric sequences.

CREATING TIME SERIES AND SIGNALS

A series, as we will use the word in this chapter, is a collection of *data* observations associated with ordered *positions*, which can be in the time, space, or frequency domain, but must be monotonic (though not strictly monotonic). If the positions of the series correspond to the calendar dates or the time of day of the data observations, then we refer to it as a calendar-based series, or time series. If the positions of the series correspond to elapsed time, frequency, or spatial measurement, but not a particular time of day or calendar date, then we refer to it as a non-calendar-based series, or signal. The following two sections show how to create the two types of series in Spotfire S+.

Creating Calendar-Based Time Series

Calendar-based time series are stored in Spotfire S+ in objects of class `"timeSeries"`, and created with the `timeSeries` function. In its simplest form, the `timeSeries` function takes a time/date or time sequence object as its `positions` argument, and any rectangular data object (for example, vector, matrix, or data frame) as its `data` argument:

```
> x <- timeSeries(pos = timeCalendar(d = 1:10, m = 1,
+ y = 1998, format = "%02m/%02d/%Y"),
+ data = data.frame(x = 11:20, y = 21:30))

> x

Positions   x  y
01/01/1998 11 21
01/02/1998 12 22
01/03/1998 13 23
01/04/1998 14 24
01/05/1998 15 25
01/06/1998 16 26
01/07/1998 17 27
01/08/1998 18 28
01/09/1998 19 29
01/10/1998 20 30
```

See Chapter 6, Dates, Times, Time Intervals, and Sequences, for more information on time/date and time sequence objects.

The timeSeries function can also generate a time sequence for the positions automatically. By default, the positions will be a daily sequence starting on January 1, 1960, but you can also specify different starting dates and sequence increments. For example, the following command generates a bimonthly sequence starting on January 1, 1998:

```
> timeSeries(data = 1:12, from = "1/1/1998", by = "months",
+ k.by = 2)
```

```
              Positions  1
01/01/1998 00:00:00.000  1
03/01/1998 00:00:00.000  2
05/01/1998 00:00:00.000  3
07/01/1998 00:00:00.000  4
09/01/1998 00:00:00.000  5
11/01/1998 00:00:00.000  6
01/01/1999 00:00:00.000  7
03/01/1999 00:00:00.000  8
05/01/1999 00:00:00.000  9
07/01/1999 00:00:00.000 10
09/01/1999 00:00:00.000 11
11/01/1999 00:00:00.000 12
```

Besides the data and positions, time series objects also have slots for a title, documentation, units, and other information that you might want to store with the data. This information can be added to a time series (and displayed) by accessing the slots directly:

```
> x@title <- "My Time Series"
> x@documentation <- c("This is the documentation",
+ "for this time series")
> x@units <- c("unit1", "unit2")
> x@title
```

```
[1] "My Time Series"
```

You can convert an old-style calendar-based time series (class "cts", or "its" with dates-based times) to a new time series object by calling ts.update.

```
> ts.update(cts(1:10, start = dates("1/15/1993"),
+ units = "weeks", k.units = 2))

              Positions  1
01/15/1993 00:00:00.000  1
01/29/1993 00:00:00.000  2
02/12/1993 00:00:00.000  3
02/26/1993 00:00:00.000  4
03/12/1993 00:00:00.000  5
03/26/1993 00:00:00.000  6
04/09/1993 00:00:00.000  7
04/23/1993 00:00:00.000  8
05/07/1993 00:00:00.000  9
05/21/1993 00:00:00.000 10
```

Creating Non-Calendar-Based Signals

Non-calendar-based series, or signals, are stored in Spotfire S+ in objects of class "signalSeries", and are created with the signalSeries function. In its simplest form, the signalSeries function takes a numeric vector or sequence as its positions argument, and any rectangular object (for example, a vector, matrix, or data frame) as its data argument:

```
> signalSeries(pos = 1:10, data = data.frame(x = 11:20,
+ y = 21:30))

Positions  x  y
 1        11 21
 2        12 22
 3        13 23
 4        14 24
 5        15 25
 6        16 26
 7        17 27
 8        18 28
 9        19 29
10        20 30
```

The signalSeries function also allows you to create regularly-spaced numeric positions automatically:

```
> x <- signalSeries(data = data.frame(x = 11:20), from = 1,
+ by = 1)
```

```
> x

Positions  x
 1        11
 2        12
 3        13
 4        14
 5        15
 6        16
 7        17
 8        18
 9        19
10        20
```

Besides the data and positions, signal objects also have slots for a title, documentation, units, and other information that you might want to store with the data. This information can be added to a signal (and displayed) by accessing the slots directly:

```
> x@title <- "My Signal"
> x@documentation <- c("This is the documentation",
+ "for this signal")
> x@units <- c("unit1", "unit2")
> x@units.position <- "seconds"
> x@title

[1] "My Signal"
```

You can convert an old-style non-calendar-based time series (class `"rts"`, or `"its"` with numeric times) to a new signal object by calling `ts.update`:

```
> ts.update(rts(1:12, start = c(1953, 4), frequency = 12))

Positions  1
1953.250   1
1953.333   2
1953.417   3
1953.500   4
1953.583   5
1953.667   6
1953.750   7
```

```
1953.833   8
1953.917   9
1954.000  10
1954.083  11
1954.167  12
```

SUBSETTING AND BASIC MANIPULATION OF SERIES

Time series and signals can be manipulated in Spotfire S+ in the same way as most other S-PLUS objects. Subscripting a series object always results in the same type of series object, and row and column subscripting works the same way as for matrices and data frames (a series containing only one vector column behaves like a 1-column matrix for subscripting).

You can use the standard S-PLUS mathematical and summary functions, and arithmetic and comparison operators on series as well. If you operate on a series with a non-series object, the operation will be applied to the data in the series, and the return value will generally be a series. If you operate on two series objects, both will first be expanded, if necessary inserting `NA` values, so that they have the same positions, and then the operation will be performed, so that the operation always acts on data with the same positions.

There are also functions for extracting or replacing the positions and data of series: `positions` and `seriesData`, respectively, which you can use to extract the data first if you want to perform an operation on two series without first aligning positions, or if you want to pass series data into a function without a series method. The `positions` and `seriesData` replacement functions always maintain the validity of the series objects, and are therefore safer to use than accessing the `data` or `positions` slots directly.

Some examples of basic series manipulation:

```
> x <- signalSeries(pos = 1:10, data = 11:20)
> x[3:6,]

Positions  1
3         13
4         14
5         15
6         16
```

```
> x >= 15

Positions 1
 1        F
 2        F
 3        F
 4        F
 5        T
 6        T
 7        T
 8        T
 9        T
10        T

> positions(x)

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

INTERPOLATION AND ALIGNMENT OF SERIES

The align function can be used to interpolate or align a series to new positions. In its simplest form, it takes two arguments: a series and a vector of new positions. The new positions must be compatible with the series (i.e., calendar or non-calendar), or be convertible to a compatible class. The default output of align is a new series whose positions are the new positions vector, and whose data are the rows from the input series corresponding to the new positions (or NA if the positions do not exist). For example,

```
> x <- signalSeries(pos = 1:10, data = data.frame(a = 11:20,
+ b = 21:30))
> align(x, c(3, 5.5, 8))

Positions  a  b
3.0       13 23
5.5       NA NA
8.0       18 28
```

The how argument to the align function allows you to specify a different action to take when the new positions do not exist in the input series. The options are to drop the position entirely, take the data from the position before or after the missing position, take the data from the nearest position, or to interpolate linearly:

```
> align(x, c(3, 5.5, 8), how = "interp")

Positions    a    b
3.0       13.0 23.0
5.5       15.5 25.5
8.0       18.0 28.0

> align(x, c(3, 5.5, 8), how = "before")

Positions  a  b
3.0       13 23
5.5       15 25
8.0       18 28
```

By default, the align function only considers positions “matched” if they match exactly. However, using the matchtol argument, you can allow positions that match within a tolerance value to be treated as if they matched. For example:

```
> align(x, c(3, 5.1, 5.5, 8), matchtol = 0.2)

Positions  a  b
3.0       13 23
5.1       15 25
5.5       NA NA
8.0       18 28
```

If more than one matches within tolerance, the closest-matching position will be used.

When aligning a calendar-based time series to positions with a different time zone, normally the times and dates are compared by comparing their absolute GMT times. However, you also have the option of aligning or interpolating by comparing the displayed local clock time by setting the localzone argument to TRUE. See the section Times and Dates in Spotfire S+ in Chapter 6, Dates, Times, Time Intervals, and Sequences, for more information about time zones.

The align function also has special arguments that allow a calendar-based time series to be aligned to a regular time sequence generated from the argument list, instead of to a specified vector of positions:

```
> a <- timeSeries(pos = timeCalendar(d = 1:10, h = 1:10),
+ data = data.frame(a = 11:20, b = 5 * (1:10)))
> align(a, matchtol = 1, by = "days", k.by = 2)

               Positions  a  b
 01/01/1960 00:00:00.000 11  5
 01/03/1960 00:00:00.000 13 15
 01/05/1960 00:00:00.000 15 25
 01/07/1960 00:00:00.000 17 35
 01/09/1960 00:00:00.000 19 45
 01/11/1960 00:00:00.000 20 50
```

MERGING SERIES

The `seriesMerge` function can be used to take the union or intersection of two or more series of the same class (calendar or non-calendar). By default, `seriesMerge` takes the intersection of the input series, keeping positions only if all of the input series have them, and including data from all of the input series' columns in the order of the arguments:

```
> x <- timeSeries(pos = timeCalendar(d = 1:10, y = 1998,
+ format = "%m/%02d/%Y"), data = data.frame(a = 1:10))

> y <- timeSeries(pos = timeCalendar(d = 7:12, y = 1998,
+ format = "%m/%02d/%Y"), data = data.frame(b = 11:16))

> seriesMerge(x, y)

Positions   a  b
1/07/1998   7 11
1/08/1998   8 12
1/09/1998   9 13
1/10/1998 10 14
```

The `seriesMerge` function can also be used to take the union of two or more series, including positions if any of the input series have them, and putting `NA` in missing data cells. To do this, specify `"union"` as the `pos` argument to `seriesMerge`:

```
> seriesMerge(x, y, pos = "union")

Positions   a  b
1/01/1998   1 NA
1/02/1998   2 NA
1/03/1998   3 NA
1/04/1998   4 NA
1/05/1998   5 NA
1/06/1998   6 NA
1/07/1998   7 11
1/08/1998   8 12
1/09/1998   9 13
1/10/1998 10 14
1/11/1998 NA 15
1/12/1998 NA 16
```

The `seriesMerge` function takes the same `how`, `matchtol`, and `localzone` arguments as the `align` function (see the previous section). These arguments govern how position matching is determined and what data to include if positions do not match. Also, instead of using the intersection or union of the input series' positions, you can merge series by passing in specific positions in the `pos` argument and these will be used as the output positions. All of this facilitates very flexible series merging. For example, here is a command that merges two series, using the positions of the second for the output, and using the value before when those positions do not exist in the first series:

```
> seriesMerge(x, y, pos = positions(y), how = "before")

Positions  a  b
1/07/1998  7 11
1/08/1998  8 12
1/09/1998  9 13
1/10/1998 10 14
1/11/1998 10 15
1/12/1998 10 16
```

AGGREGATING AND COARSENING SERIES

Aggregation is a series summarization technique that allows you to take a series with a small position increment and turn it into a coarser series. For example, if you have a daily series giving production output, you might want to use that series to calculate monthly production numbers by summing the daily numbers for each month. This can be done using the `aggregate` function:

```
> x <- timeSeries(data = abs(rnorm(365)),
+ from =  timeDate("1/1/1998", format = "%02m/%d/%Y"))

> aggregate(x, by = "months", FUN = sum)

 Positions        1
 01/1/1998 25.96189
 02/1/1998 22.42384
 03/1/1998 32.35661
 04/1/1998 20.46768
 05/1/1998 25.89401
 06/1/1998 28.85333
 07/1/1998 22.12951
 08/1/1998 26.86349
 09/1/1998 28.09069
 10/1/1998 26.93640
 11/1/1998 18.17815
 12/1/1998 31.58526
```

The `aggregate` method for time series, `aggregateSeries`, can also be called directly. The arguments that we discuss in this section are documented in the online help for `aggregateSeries`.

The series `aggregate` function takes any summarizing function as its `FUN` argument; besides the `sum` function, other common choices are `mean`, `median`, and a special function `hloc` that finds the high, low, open, and close values for financial data. The intervals for aggregation can be specified using the `by` argument (for calendar-based time series only) as in the example above. Alternatively, the intervals can be specified by giving the break positions directly, as illustrated in the example below.

```
> aggregate(x, pos = timeCalendar(m = 1:12, y = 1998,
+ format = positions(x)@format), FUN = hloc,
+ colnames = c("high", "low", "open", "close"))
```

```
 Positions      high         low       open     close
 01/1/1998 2.311537 0.003693723 1.59952744 0.9121383
 02/1/1998 1.845184 0.052118065 1.56433635 0.2658049
 03/1/1998 3.167683 0.025329661 0.03012198 0.1842446
 04/1/1998 2.016290 0.027755647 1.83146372 0.2721489
 05/1/1998 3.074278 0.060317966 0.71533852 0.5911421
 06/1/1998 3.314549 0.055421084 1.32587865 0.8268578
 07/1/1998 1.883933 0.039954801 0.08814959 0.2040930
 08/1/1998 2.388555 0.112880242 1.39607369 0.8352501
 09/1/1998 2.922205 0.052163112 0.29260770 0.7463595
 10/1/1998 2.310027 0.031709100 0.03170910 2.3100267
 11/1/1998 1.831786 0.006232657 1.30919193 0.3495212
 12/1/1998 2.166783 0.079449340 1.81978978 0.2861171
```

The series `aggregate` function can also use moving sample windows instead of disjoint sets of observations; this is commonly used to calculate moving averages with the `mean` function. Another useful feature, especially in the case of moving averages, is that the positions of the output series can be adjusted to anywhere within the sample window, by specifying a value for the `adj` argument between `0` and `1`. For example, to take a financial time series and calculate the moving average over the previous twenty trading days, you might want to output positions at the end, rather than the beginning, of the sampling window:

```
> aggregate(x[1:30,], moving = 20, adj = 1, FUN = mean)
```

```
  Positions         1
 01/20/1998 0.8427219
 01/21/1998 0.7775997
 01/22/1998 0.6933231
 01/23/1998 0.7256408
 01/24/1998 0.7149336
 01/25/1998 0.7968545
 01/26/1998 0.7806625
 01/27/1998 0.8288284
 01/28/1998 0.7302111
 01/29/1998 0.7309572
 01/30/1998 0.8124860
```

PLOTTING TIME SERIES

Calendar-based time series can be plotted using the `plot` function. The `plot` method for time series is called `plot.timeSeries`, and can also be called directly with the plotting arguments documented in its help file. The `plot.timeSeries` function is flexible enough to create presentation-quality plots for most time series data, including those with observation times separated by fractions of a second or multiple years, regularly or irregularly spaced.

For a simple line plot of a single time series `x`, just call `plot(x)`. For more complicated plots, you can use the examples in this section as a guide. The examples include:

- Daily high/low/open/close plot of financial data;
- Moving average plot;
- Plot of intraday trading data spanning several days;
- Multiple related time series on the same plot;
- Plots using Trellis graphics.

Note

For best results, you should use the `trellis.device` command to set up your plotting device before calling `plot`. This ensures that you get the trellis style and color maps, which produce better-looking plots without further specifying plot parameters such as line style and colors.

High/Low/Open/Close Plot

One application of the `plot` function is to make daily high/low/open/close plots of financial data. For example, we can look at the stock market correction of October, 1987 by plotting a portion of the `djia` data set:

```
> dow <- djia[positions(djia) >= timeDate("09/01/87") &
+ positions(djia) <= timeDate("11/01/87"), 1:4]
> plot(dow, plot.type = "hloc")
```

The plot is shown in Figure 5.1. If you prefer candlestick-style to line-style indicators for high/low/open/close plots, you can use the following call instead.

```
> plot(dow, plot.type = "hloc", plot.args = list(
+ style = "c"))
```

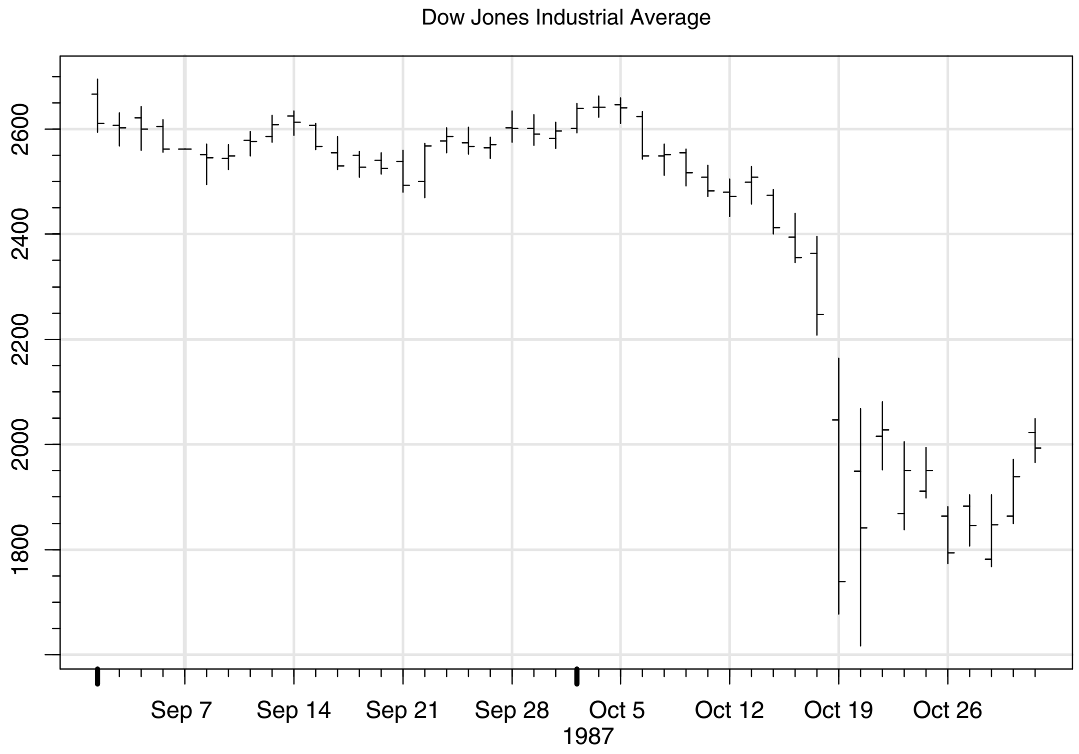


Figure 5.1: *Daily high/low/open/close plot of the Dow Jones Industrial Average during the period surrounding the stock market crash of 1987.*

Moving Average Plot

Another common financial time series plotting application is to create a moving average plot. To illustrate this, we first use the `aggregate` function (see the section Aggregating and Coarsening Series) to create a daily high/low/open/close series from the intraday trading data in the `tbond` data set:

```
> tb.hloc <- aggregate(tbond,
+ pos = timeSeq(from = timeDate("1/7/1994"),
+ to = timeDate("2/4/1995"), by = "days"),
+ colnames = c("high", "low", "open", "close"),
+ FUN = hloc, together = T)
```

Then we use the aggregate function again to create a 20-business-day moving average of the closing prices:

```
> tb.avg <- aggregate(tb.hloc[,"close"], moving = 20,
+ FUN = mean, adj = 1)
```

We then plot the high/low/open/close series by calling the plot function, and then add the moving average line to the plot by calling the lines.render function with the scale calculated by plot:

```
> plot.out <- plot(tb.hloc, plot.type = "hloc",
+ main = "T-Bonds")
> lines.render(positions(tb.avg), seriesData(tb.avg),
+ x.scale = plot.out$scale)
```

This plot is shown in Figure 5.2.

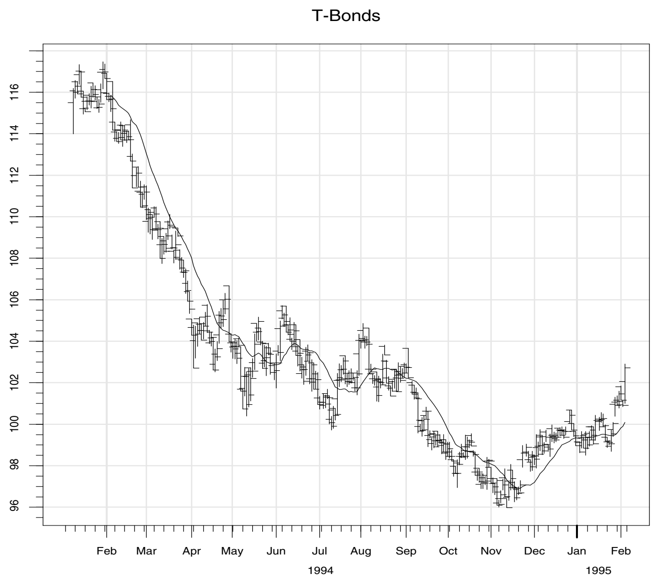


Figure 5.2: *Treasury bill futures, daily high, low, open, and close, with 20-day moving average superimposed.*

Intraday Trading Data Plot

Another application of the `plot` function is to plot intraday trading data that spans a few days. For example, the `tbond` data set used in the last example has high and low prices every twenty minutes over its time span for treasury bond futures trading. We can look at two weeks of this trading data by calling:

```
> tb <- tbond[positions(tbond) >= timeDate("02/01/94") &
+ positions(tbond) <= timeDate("02/13/94"),]

> plot(tb, plot.type = "hloc")
```

This plot is shown in Figure 5.3. We use the `"hloc"` plot type, even though there are only high and low price values, to draw a line from the low to high values for each time. The `plot` function automatically puts in an axis break for each night when the market is closed.

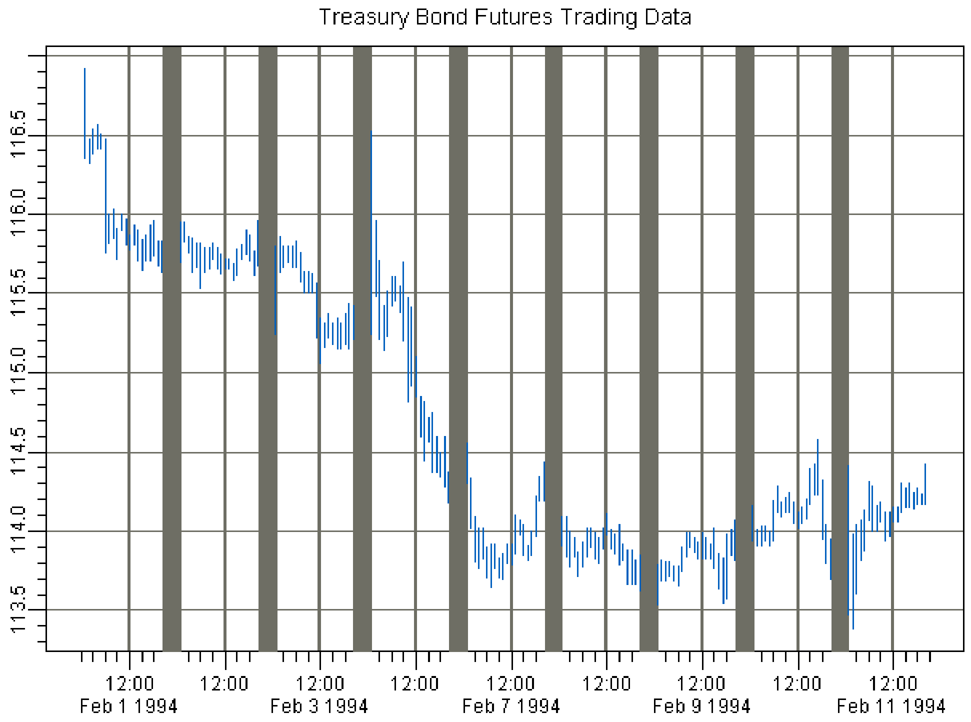


Figure 5.3: *Two weeks of trading data for treasury bond futures, showing the high and low trading prices every twenty minutes during trading hours.*

Plots Containing Multiple Time Series

Another application of the plot function is to plot multiple related curves on the same plot. For example, the tbauc data sets contain interest rates for auctions of treasury bills of various maturities. We can compare them by plotting:

```
> tb3m <- tbauc.3m[positions(tbauc.3m) >=
+ timeDate("01/01/96") & positions(tbauc.3m) <=
+ timeDate("07/01/97"),]

> tb6m <- tbauc.6m[positions(tbauc.6m) >=
+ timeDate("01/01/96") & positions(tbauc.6m) <=
+ timeDate("07/01/97"),]

> tb1y <- tbauc.1y[positions(tbauc.1y) >=
+ timeDate("01/01/96") & positions(tbauc.1y) <=
+ timeDate("07/01/97"),]

> plot(tb3m, tb6m, tb1y)
```

The plotting function calls seriesMerge (see the section Merging Series) to merge the three passed-in time series. By default, seriesMerge would create the intersection of the three time series, but the plotting function passes in additional arguments from its merge.args argument. The defaults in merge.args create a union of the passed-in series, and interpolate each series for times they do not share, which makes nice-looking line plots for series that do not share times, as seen in the output in Figure 5.4.

It is also possible to merge the series in different ways to produce different plots of the three series, by putting different arguments for seriesMerge in the plot function call. For example, if you want to avoid interpolating and just view the raw data, you could take the union of the series but insert NA for times the series do not share, and then plot with points instead of lines:

```
> plot(tb3m, tb6m, tb1y, merge.args = list(pos = "union"),
+ plot.args = list(type = "p"))
```

The output of this plot is shown in Figure 5.5.

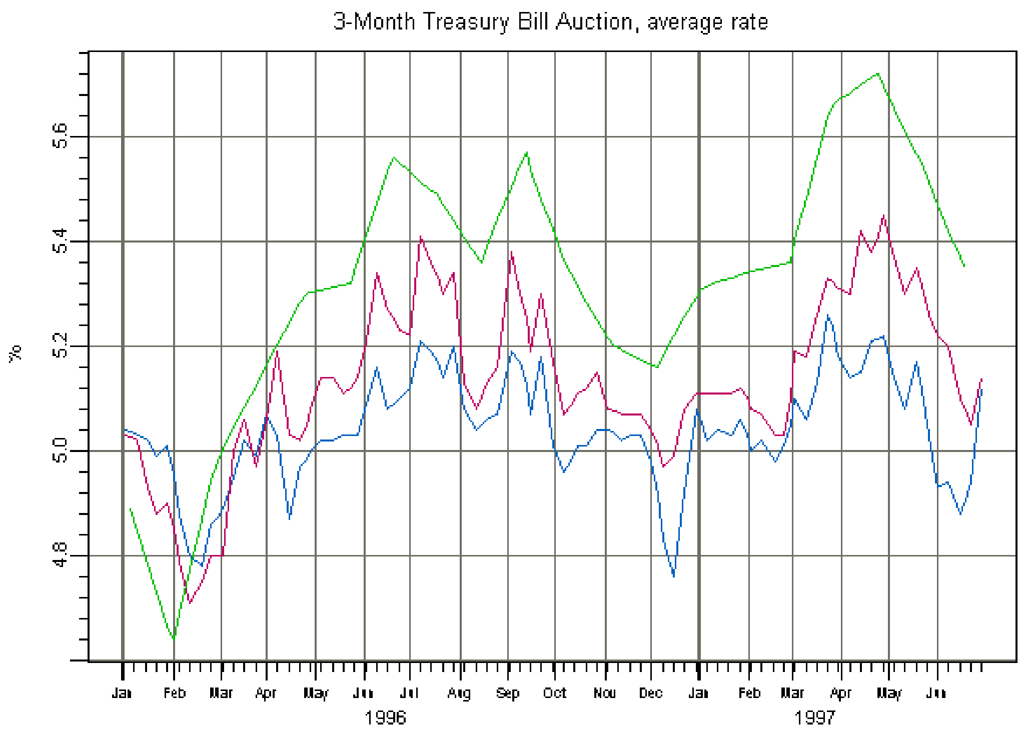


Figure 5.4: *Treasury bond auction rates for three different maturities of treasury bonds, using the default plotting arguments.*

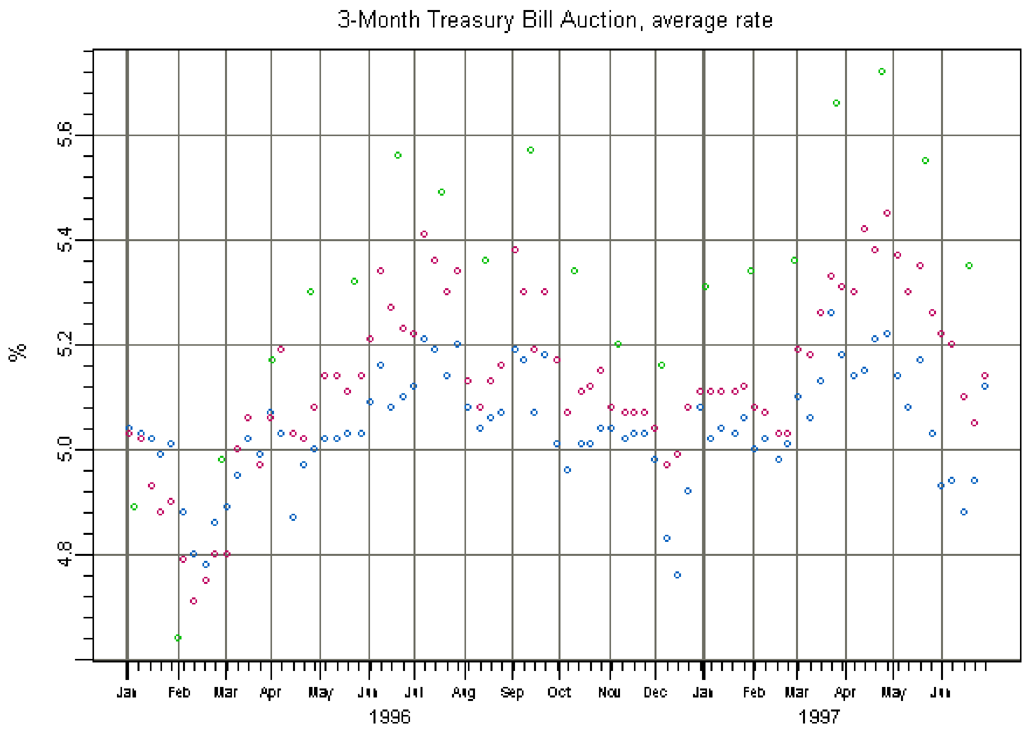


Figure 5.5: *Treasury bond auction rates for three different maturities of treasury bonds, plotting raw data without interpolation.*

Time Series Trellis Plots

While the `plot` function used in the previous examples is sufficient for many purposes, Trellis graphics offers two main advantages over basic Spotfire S+ plotting:

- The production of split-panel plots with header bars on each panel;
- The ability to create output objects on one plotting device and print them on another, resulting in attractive plots on both devices that have sensible default colors, shading, and line types.

In this section, we demonstrate how to make Trellis plots of time series.

The basic Trellis plotting function for time series is the `trellisPlot` function. The `trellisPlot` method for time series is called `trellisPlot.timeSeries`, and can also be called directly with the plotting arguments documented in its help file. Here is an example that shows how to create a Trellis plot of part of the `djia` data set, view it on the screen in color, and then send the same output to a postscript file:

```
> djia1 <-djia[positions(djia) >= timeDate("09/01/87") &
+ positions(djia) <= timeDate("11/01/87"), 1:4]

> trellis.device()
> plot.obj <- trellisPlot(djia1)
> print(plot.obj)
> trellis.device("postscript", file = "out.ps")
> print(plot.obj)
> dev.off()
```

The output is shown in Figure 5.6.

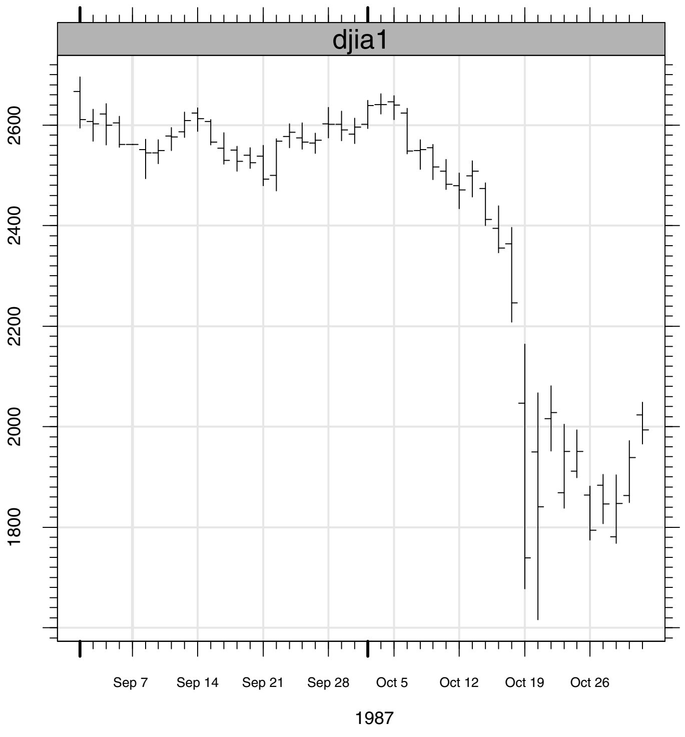


Figure 5.6: *Trellis plot of the Dow Jones Industrial Average.*

The djia data set also contains trading volume information. One way to look at the price along with the trading volume is to use a split-panel Trellis plot with different-scaled *y*-axes for the price and volume panels. Here is how to create such a plot:

```
> dow <-djia[positions(djia) >= timeDate("09/01/87") &
+ positions(djia) <= timeDate("11/01/87"),]
> trellis.device()
> trellisPlot(dow[,5], dow[,1:4],
+ plot.type = list("stackbar","hloc"), layout = c(1,2),
+ scales = list(y = list(relation = "free")))
```

The output of this plot is shown in Figure 5.7. Note that trellisPlot plots each series argument on a separate panel, unlike plot. If you want to plot multiple series on the same panel, you must merge them first; see the section Merging Series for more details.

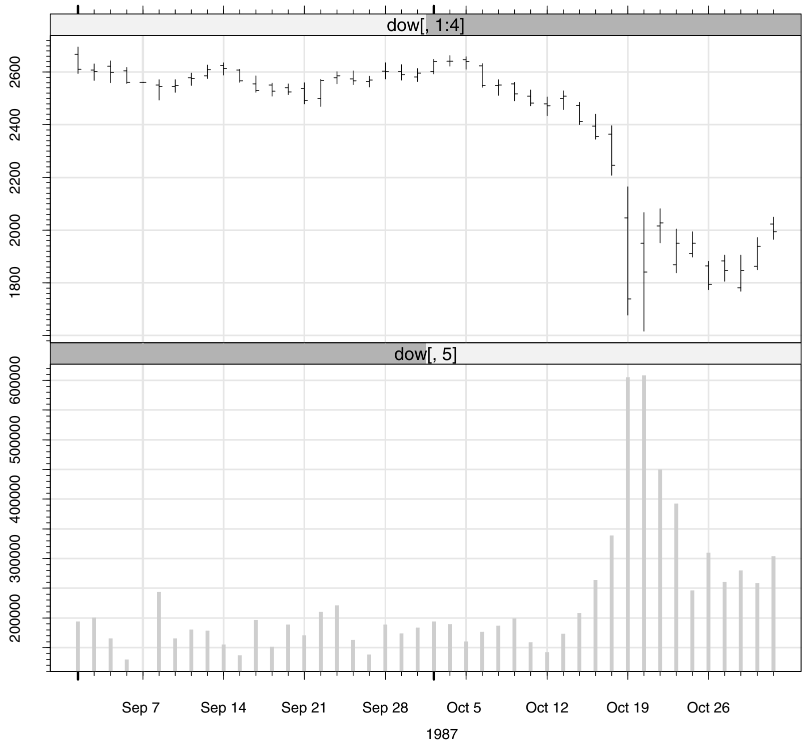


Figure 5.7: *Split-panel Trellis plot of the Dow Jones Industrial Average and its trading volume.*

Customizing Time Series and Signal Plots

There are several ways that time series and signal plots can be customized. The basic (non-Trellis) time series plots can be customized by passing additional arguments to the `plot` function; see the help files for `plot.timeSeries` and `plot.signalSeries` for more information. The most common arguments are `main`, `xlab`, and `ylab` to change the main plot title and axis labels, and `plot.type` to change how the plotted points are rendered (lines, stacked bars, etc.). The Trellis plotting function `trellisPlot` has similar arguments that allow you to customize the look of the plot, and in addition you can pass in the regular Trellis arguments such as `scales`, `main` and `ylab`; see the help files for `trellisPlot.timeSeries`, `trellisPlot.signalSeries`, and `trellis.args` for more information. The sample plots in the previous sections show several examples of these types of customization.

In addition, parameters such as colors, line widths, line types, and styles can be customized in the Trellis settings data sets. See Chapter 3, Traditional Trellis Graphics, in the *Guide to Graphics*, and the help files for `trellis.settings` and `trellis.par.set` for more information. Parameters specific to time series plotting include parameters for the tick marks (`small.tick`, `medium.tick`, and `big.tick`), parameters for the tick labels (`main.label` and `outer.label`), parameters for breaks in the time axis (`breaks`), parameters for reference grids in the plot region (`minor.grid` and `major.grid`), and data plotting style parameters (`style.type`, `hloc.line`, `hloc.candle`, and `stackbar`).

PLOTTING SIGNALS

Like calendar-based time series, signals and non-calendar-based time series can be plotted using the plot function. The plot method for signals is called plot.signalSeries, and can also be called directly with the plotting arguments documented in its help file. In this section, we show how to create plots of signals with regular and Trellis graphics. Most of the suggestions in the section Plotting Time Series also apply to signals, and this information is not repeated here.

Note

For best results, you should use the trellis.device command to set up your plotting device before calling plot. This ensures that you get the trellis style and color maps, which produce better-looking plots without further specifying plot parameters such as line style and colors.

Basic Signal Plotting

Simple signal plots can be generated by calling the plot function, which plots the signal's *y*-values against its positions. For example, plot the say.wavelet speech signal data set by calling:

```
> plot(say.wavelet)
```

The plot function also allows you to specify logarithmic axes (which would not be very appropriate here), labels for the axes, a main title, and other options. By default, the labels are taken from the series; see the section Customizing Time Series and Signal Plots. For instance, we can add labels to the simple plot:

```
> plot(say.wavelet, main = "Speech Signal",
+ ylab = "Amplitude")
```

The output is shown in Figure 5.8.

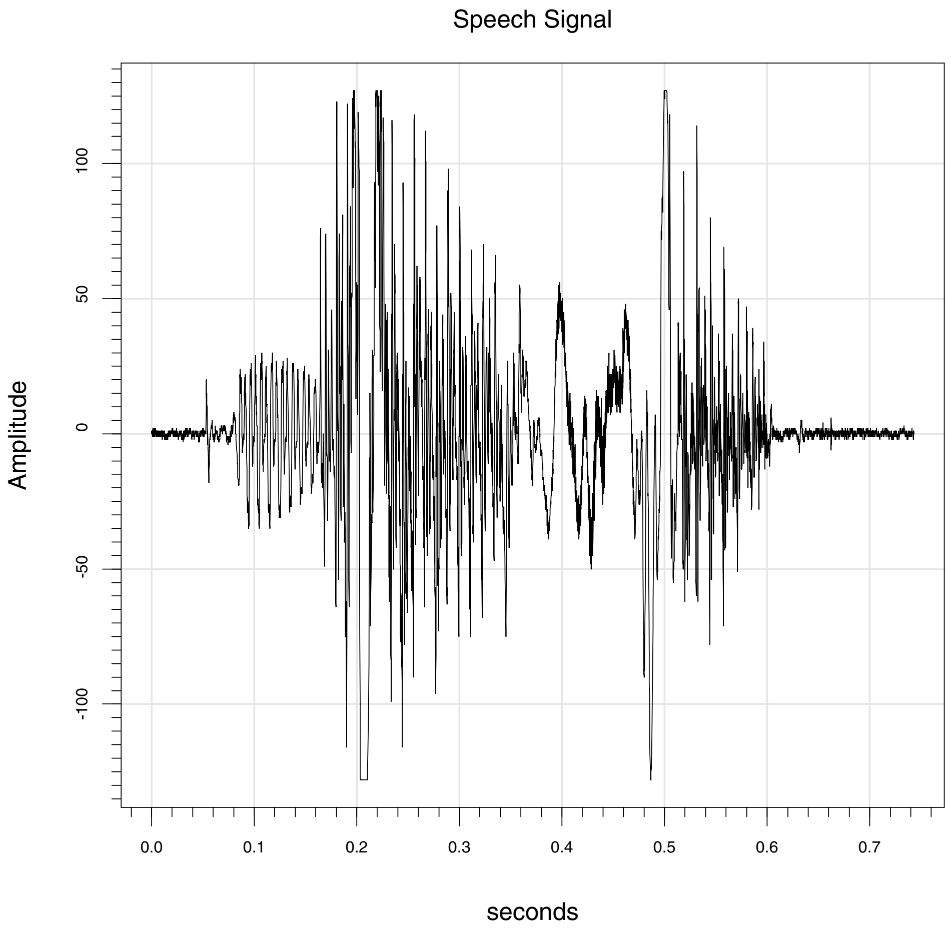


Figure 5.8: *Signal plot of a voice saying "wavelet."*

Signals with complex values must be converted to real numbers before plotting. By default, the `plot` function uses the `Mod` function to take the modulus of complex numbers. However, you can supply a different function if you wish by using the `complex.convert` argument to `plot`. Also, you can make a dB plot of a signal representing a spectrum by setting the `dB` argument to True, and semi-log or log-log plots by setting the `log.axes` argument to `"x"`, `"xy"`, or `"y"`.

Multiple signals and multivariate signals can also be plotted using the `plot` function, just as for time series. See the section Plots Containing Multiple Time Series.

Trellis Plots of Signals

While the `plot` function used in the signal plotting example above is sufficient for many purposes, Trellis graphics offers two main advantages over basic Spotfire S+ plotting:

- The production of split-panel plots with header bars on each panel, including separate panels for real and imaginary parts or phase and magnitude, if desired;
- The ability to create a plotting output object on one device and print it on another, resulting in attractive plots on both devices that have sensible default colors, shading, and line types.

In this section, we demonstrate how to make Trellis plots of signals.

The basic Trellis plotting function for signals is the `trellisPlot` function. The `trellisPlot` method for signals is called `trellisPlot.signalSeries`, and can also be called directly with the plotting arguments documented in its help file. For instance, to make a Trellis plot of the `say.wavelet` speech signal:

```
> trellisPlot(say.wavelet)
```

This plot is very similar to the basic non-Trellis plots we saw in the last section, and like the non-Trellis plots, you can use the `log.axes` and `dB` arguments to make semi-log, log-log, or dB plots.

For complex signals, Trellis plotting provides much more flexibility than the basic plotting function. To explore this, we add an imaginary component to the `say.wavelet` speech signal:

```
> cplx.spch <- say.wavelet * exp(-360i * pi *
+ positions(say.wavelet))
```

The basic plotting function allows us to plot either the modulus, real part, imaginary plot, or phase of this signal; using Trellis graphics, we can plot all four in separate panels with separate *y*-axis scales. The following command produces the Trellis graph shown in Figure 5.9. We plot the phase portion of the new signal with a small plotting symbol so that we can see patterns. We plot the other components of the signal using lines by default demodulation at 180 Hz.

```
> trellisPlot(cplx.spch, polar = T, real.im = T,
+ scales = list(y = list(relation = "free")), pch = ".")
```

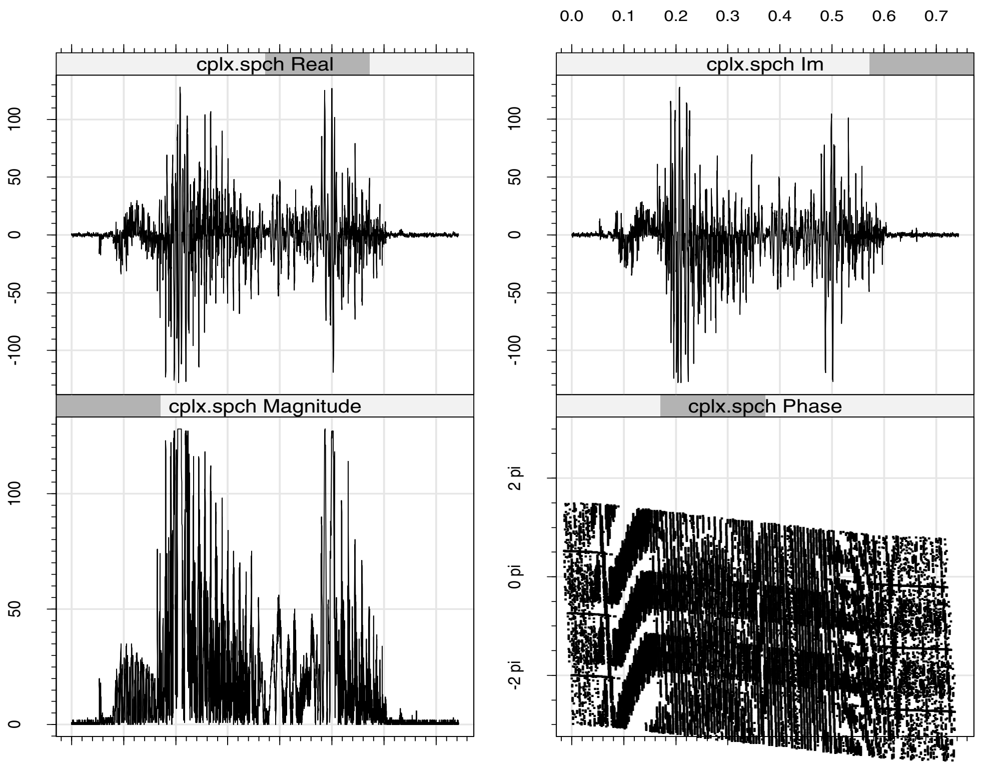


Figure 5.9: *Complex signal Trellis plot.*

DATES, TIMES, TIME INTERVALS, AND SEQUENCES 6

INTRODUCTION

In many data analysis applications, some aspect of the data is related to dates and/or times of the day. For instance, most economic data are related to the calendar, and analysis involving multiple economic data sets usually involves combining data pertaining to the same period. Financial trading data typically match a traded price and quantity with the time of day of the trade, and the calculation of investment returns and other analyses depend on the time or date of the trade. In these and countless other examples, numeric or categorical data correspond to particular dates or times of day; this type of data is known as *time series* data, and its analysis is discussed in detail in Chapter 5, Time Series and Signal Basics, and Chapter 27 of the *Guide to Statistics, Volume 2*, Analyzing Time Series and Signals. Separate from the analysis of time series, in this chapter we discuss the underlying date and time data itself and related issues, such as representing time intervals, regular sequences, and one-time or recurring events in Spotfire S+.

TIMES AND DATES IN SPOTFIRE S+

Times and dates in Spotfire S+ are stored in objects of class `"timeDate"`, which, besides storing the date and time of day to the nearest millisecond, also store a display format and time zone. The following sections show how to use time/date objects in Spotfire S+, including how to create time/date objects from character data, display times and dates, create time/date objects from numeric data, use time zones, calculate holiday dates, and perform basic subsetting and manipulation of time/date objects.

Creating Time/Date Objects from Character Data

The `timeDate` and `as` functions can be used to read times and/or dates from character strings using the default input format:

```
> as(c("1/1/97", "2/1/97", "mar 1, 1997",
+ "April 1, 1997 3PM"), "timeDate")
```

```
[1] 01/01/1997 00:00:00.000 02/01/1997 00:00:00.000
[3] 03/01/1997 00:00:00.000 04/01/1997 15:00:00.000
```

```
> timeDate(c("1/1/97", "2/1/97", "mar 1, 1997",
+ "April 1, 1997 3PM"))
```

```
[1] 01/01/1997 00:00:00.000 02/01/1997 00:00:00.000
[3] 03/01/1997 00:00:00.000 04/01/1997 15:00:00.000
```

As you can see, when `x` is a character vector, `timeDate(x)` and `as(x,"timeDate")` produce the same output, and the default input format is somewhat flexible. The `timeDate` function also allows you to specify your own input format using the `in.format` argument:

```
> timeDate(c("1 Jan 1992 skip 2", "1 Jan 1992 skip 3",
+ "1 Jan 1992 skip 4"), in.format = "%d %m %y %w %H")
```

```
[1] 01/01/1992 02:00:00.000 01/01/1992 03:00:00.000
[3] 01/01/1992 04:00:00.000
```

```
> timeDate(c("1 PM", "2 PM", "3 AM"), in.format = "%H %p")
```

```
[1] 01/01/1960 13:00:00.000 01/01/1960 14:00:00.000
[3] 01/01/1960 03:00:00.000
```

```
> timeDate(c("3/1/1992", "15/5/1998"),
+ in.format = "%d/%m/%y", format = "%d/%m/%y")

[1] 3/1/92  15/5/98
```

Input formats are single-element character vectors consisting of input fields (which start with "%" and end with a letter) and other characters (such as letters, ":", ".", "/") that must be matched exactly. Commonly used input specifications include:

`%d`	Input day of month as integer (1-31)
`%m`	Input month as integer (1-12) or character string
`%y`	Input year as integer; add current century if 2-digit
`%H`	Input hour of day as integer (0-23)
`%M`	Input minute of day as integer (0-59)
`%S`	Input second of day as integer (0-59)
`%N`	Input tenths, hundredths, or thousandths of a second as if following decimal point
`%Z`	Input time zone name
`%#c`	Skip # characters
`%$c`	Skip rest of input string
`%w`	Skip one word

Notes

The current century for the `%y` specification is controlled by `options("time.century")`. The default setting is 1930, which causes 30-99 to be interpreted as 1930-1999 and 00-29 as 2000-2029.

The month names and abbreviations are controlled by `options("time.month.name")` and `options("time.month.abb")`. They can be changed to use languages other than English for dates.

The default input format is stored in `options("time.in.format")`. In Microsoft Windows®, this is set according to Windows Regional Settings. To set the option so that it is equivalent to the default values in Spotfire S+ for UNIX®, add the following expression to your **S.init** file:

```
options(time.in.format = "[%m[/][.]%d[/][,]%y][%H[:%M[:%S[.%N]]][%p][[(]%3Z[)]]]",
        time.out.format = "%02m/%02d/%Y %02H:%02M:%02S.%03N",
          time.out.format.notime = "%02m/%02d/%Y")
```

There is also a list of sample formats in `format.timeDate`.

See the help document on the `timeDate` class for a complete description of time input formats.

Displaying Time/Date Objects

Time/date objects have a default display and printing format, as seen in the previous section, that displays both the date and time of day in a standard, fixed-width format. You can change the output display format of a time/date object, for instance, to display just the time, display just the date, print the day of the week, or use different separation characters. One way to do this is to include a `format` argument in the `timeDate` function when you create the time/date object:

```
> timeDate(c("1 PM", "2 PM", "3 AM"), in.format = "%H %p",
+ format = "%02H:%02M")

[1] 13:00 14:00 03:00
```

You can also change (or view) the format on an existing time/date object by accessing the `format` slot:

```
> x <- timeDate(c("1/3/1998", "2/5/2014", "10/17/1857"))
> x@format = "%2m/%02d/%y"
> x@format
```

```
[1] "%2m/%02d/%y"

> x

[1]  1/03/98    2/05/14   10/17/1857

> x@format = "%2d/%02m/%y"

> x

[1]  3/01/98    5/02/14   17/10/1857
```

Output formats are single-element character vectors consisting of output fields (which start with "%" and end with a letter) and other characters (such as letters, ":", ".", "/") that are simply printed. Commonly used output specifications are listed in the table below.

%d	Print day of month as integer (1-31)
%a	Print weekday abbreviation
%A	Print weekday full name
%m	Print month as integer (1-12)
%b	Print month abbreviation
%B	Print month name
%Y	Print 4-digit year
%C	Print 2-digit year, subtracting off the century
%y	Print 2-digit year if within current century, otherwise 4-digit year
%H	Print hour of day as integer (0-23)
%I	Print hour in 12-hour clock format as integer (1-12)
%p	Print AM or PM string
%M	Print minute of day as integer (0-59)

`%S`	Print second of day as integer (0-59)
`%03N`	Print decimal fraction of second, using 3 decimal places (decimal point not included)
`%z`	Print time zone name

Notes

To make an integer field print with a fixed width, put a number representing your desired width between the `%` and the output code. For example, if the day of the month is 7 and it is printed with `%d`, it will appear as "7"; if it is printed with `%2d`, it will appear as " 7" with a space before the number. You can also pad with zeros instead of spaces, by putting a zero before the field width, for example, `%02d` to print as "07". The field width must be large enough to hold the entire number, or "*" asterisk characters will be inserted in the field to indicate overflow. Character fields can also be made fixed-width in the same way: the leftmost characters are printed if the field is too short, and extra spaces are added if it's too long. Padding with zeros is not allowed for character fields, and the field width can be shorter than the character string to make abbreviations.

The current century for the `%y` specification is controlled by `options("time.century")`. The default setting is 1930, which causes 1930-2029 to be printed in two digits, and everything else in four digits.

The day and month names and abbreviations are controlled by `options("time.day.name")`, `options("time.day.abb")`, `options("time.month.name")`, and `options("time.month.abb")`.

The default output format is stored in `options("time.out.format")`. In Microsoft Windows, this is set according to Windows Regional Settings. To set the option so that it is equivalent to the default values in Spotfire S+ for UNIX, add the following expression to your **S.init** file:

```
options(time.in.format = "[%m[/][.]%d[/][,]%y][%H[:%M[:%S[.%N]]][%p][[(]%3Z[)]]]",
        time.out.format = "%02m/%02d/%Y %02H:%02M:%02S.%03N",
          time.out.format.notime = "%02m/%02d/%Y")
```

There is also a list of sample formats in `format.timeDate`. The output format in `options("time.out.format.notime")` is used for time/date objects with date information only; if this option is not defined, the default format is used instead.

See the help document on the `timeDate` class for a complete description of time output formats.

Creating Time/Date Objects from Numeric Data

There are several types of numeric data that can be converted to time/date objects. One possible scenario is that you know the month, day, year, hour of the day, minute, and/or second, and you want to convert that information into a time/date object. This is done using the `timeCalendar` function, as illustrated in the following commands:

```
> timeCalendar(m=3, d=1:10, y=1998, format="%d/%m/%y")

[1] 1/3/98  2/3/98  3/3/98  4/3/98  5/3/98  6/3/98
[6] 7/3/98  8/3/98  9/3/98  10/3/98

> timeCalendar(h=0:23, format="%02H:%02M")

 [1] 00:00 01:00 02:00 03:00 04:00 05:00 06:00 07:00
 [9] 08:00 09:00 10:00 11:00 12:00 13:00 14:00 15:00
[17] 16:00 17:00 18:00 19:00 20:00 21:00 22:00 23:00
```

Another type of numeric data that can easily be converted to a time/date object is numeric data representing the number of days since some date origin or representing the time of day as an elapsed time. The easiest data to convert represent the date and time as the number of days (possibly including fractional days) since the standard S-PLUS date origin of January 1, 1960. This type of data can be converted to a time/date object using the `as` function:

```
> as(c(-1, 0, 1, 1.5), "timeDate")

[1] 12/31/1959 00:00:00.000 01/01/1960 00:00:00.000
[3] 01/02/1960 00:00:00.000 01/02/1960 12:00:00.000
```

If you have numeric data representing the number of days since a different date origin, you can either first subtract off the number of days between your date origin and January 1, 1960 or use the `timeDate` function's `julian` and `in.origin` arguments:

```
> timeDate(julian = 1:10, in.origin = c(month=1, day=1,
+ year=1998))

 [1] 01/02/1998 01/03/1998 01/04/1998 01/05/1998 01/06/1998
 [6] 01/07/1998 01/08/1998 01/09/1998 01/10/1998 01/11/1998
```

Basic Operations on Time/Date Objects

Time/date objects can, in most cases, be manipulated like most other S-PLUS objects. For instance, subscripting works the same as for other vectors, and you can use the `length` function to count or set the number of elements. You can concatenate time/date objects using the `concat` function, and you can convert them to character, numeric, and integer vectors using the `as` function.

Various mathematical operations are also defined for time/date objects. Numbers can be added to or subtracted from time/date objects: The integer part is added to or subtracted from the date and the fractional part to the time of day, carrying over into the date as necessary. Subtraction of two time/date objects (or using the `diff` function) results in a time span object; see section Time Intervals in Spotfire S+. Time spans and other time interval objects can be added to or subtracted from time/date objects. Time/date objects can be compared with each other and with numbers using the usual comparison operators. Some examples:

```
> x <- timeDate(c("1/1/1998 2:00", "1/5/1998 15:00"))
> y <- timeDate(c("1/1/1998 3:00", "1/5/1998 12:00"))
> x + 1.5

[1] 01/02/1998 14:00:00.000 01/07/1998 03:00:00.000

> x < y

[1] T F
```

Many other basic S-PLUS vector manipulation functions also work on time/date objects; `mean`, `quantile`, `max`, `floor`, `sort`, `rev`, `match`, `unique`, and `cut` are some examples. There are also some special functions that operate only on time/date objects, such as `days`, `months`, `hours`, and `mdy`. Operations which do not make sense for times and dates cannot be performed (for example, adding two time/date objects, taking the logarithm, or multiplying or dividing a time/date object by a number).

Calculating Holiday Dates

There are several ways to calculate the dates of holidays in Spotfire S+. First, many holidays, including New Year's Day, Christmas, and Australia Day, have built-in functions to calculate them. For the complete list, see the help document for `holiday.Christmas`. You can either call these functions directly or use the `holidays` function to calculate a sorted vector of multiple holidays.

For example:

```
> holiday.NewYears(1998:2001)

[1] 01/01/1998 01/01/1999 01/01/2000 01/01/2001

> holidays(1998:2001, c("Christmas", "NewYears"))

[1] 01/01/1998 12/25/1998 01/01/1999 12/25/1999 01/01/2000
[6] 12/25/2000 01/01/2001 12/25/2001
```

There are also several S-PLUS functions you can use to calculate dates of holidays that are not built in or to write your own holiday functions. The `holiday.fixed` function calculates the date of a holiday that falls on the same date every year. The `holiday.weekday.number` function calculates the date for a holiday that falls, for example, on the third Monday in May or the last Thursday in October every year. The `holiday.nearest.weekday` function moves a holiday that falls on a weekend to the nearest weekday. Here are some examples:

```
# Christmas
> holiday.fixed(1998:2001, 12, 25)

[1] 12/25/1998 12/25/1999 12/25/2000 12/25/2001

# Memorial Day (last Monday in May)
> holiday.weekday.number(1998:2001, 5, 1, -1)

[1] 05/25/1998 05/31/1999 05/29/2000 05/28/2001

# Weekday nearest to Christmas
> holiday.nearest.weekday(holiday.fixed(1998:2001, 12, 25))

[1] 12/25/1998 12/24/1999 12/25/2000 12/25/2001
```

Tip

If you define a new holiday-calculating function that takes a vector of years as its one argument and you name it, for example, `holiday.MyNewDay`, then you will be able to access it through the `holidays` function by calling `holidays(years, c("Christmas", "MyNewDay"))`, just like the built-in holidays.

Using Time Zones

Many users will be able to completely ignore time zones in their data analysis involving times and dates, as we have done up to this point in the chapter. However, time zones do play a role in some tasks. For example, in analyzing how world financial markets react to various types of events, it is important to know when the news of each event reached each market, in each market's local time zone. In addition, in an analysis involving calculations of time intervals, you should take into consideration whether the time changes to or from daylight savings (summer) time within the data set in order to calculate the correct intervals.

Spotfire S+ provides extensive support for time zones in order to facilitate such analysis. In this section, we discuss how to define time zones and names for time zones, create time/date objects with time zones, convert times to different time zones, and perform various mathematical operations using time zones.

Built-in and Customized Time Zones

Internally, all times and dates in Spotfire S+ are stored in Greenwich Mean Time (GMT, more correctly, but perhaps less commonly, known as Universal Coordinated Time or UTC). GMT has no daylight savings time and is used as the reference for all other time zones. Spotfire S+ also has built-in support for all 24 hourly standard time zones around the planet, as well as many time zones that change to daylight savings time in the summer. Each of the built-in time zones has a fixed name; the zones and their names can be found in the help document on the `timeZone` class.

Different users around the world will want to refer to the same time zones using different names and abbreviations. At the same time, the fixed names of the built-in time zones need to be unambiguous and not customizable. To provide flexibility to the user, the following scheme is used. The user begins by defining the alias names he or she wants to use for each built-in time zone by using the `timeZoneList` function. This can include adding new alias names or redefining existing names, though it is not advisable to redefine the `GMT` alias. Spotfire S+ then uses the alias names for all time zone operations, finding the translation to built-in names from the time zone list. For instance, a user in the United States would probably want to have `EST` refer to the U.S. Eastern time zone, which is the default entry in the time zone list. However, a user in Canada would probably prefer to have `EST` refer to the Eastern time zone for Canada. The two are almost the same, but Canada and the U.S. had different daylight savings time changes during some years during the oil crisis in the 1970s, and the two built-in time zones reflect this. The Canadian user can set up this behavior by calling:

```
> old.list <- timeZoneList(EST = timeZoneC("can/eastern"))
```

You can see all the current time zone names in the time zone list by calling the `timeZoneList` function with no arguments, and the previous version of the time zone list is returned from a call, like the example above, that modifies the list.

Another customization that is useful for time zone names comes into play because in some countries, the names of time zones change depending on whether it is daylight savings time or not. For instance, the U.S. Eastern time zone is called “Eastern Daylight Time” or `EDT` when it is on daylight savings time and “Eastern Standard Time” or `EST` when it is not. Spotfire S+ has automatic support for printing changing time zone names such as these if a time zone alias of the form “XXX/YYY” is defined, where XXX is the name of the time zone in standard time and YYY is the name of the zone in daylight savings time. For example, here is an entry from the default time zone list for the U.S. Eastern time zone:

```
> timeZoneList()[["EST/EDT"]]

timeZoneC("us/eastern")
```

Finally, users can also customize time zones by creating their own zones with or without daylight savings time, instead of using the built-in time zones. User-defined time zones are stored in class `"timeZoneS"` and created with the `timeZoneS` function; like new alias names, they must also be added to the time zone list. For instance, to use a time zone for a hypothetical Island Nation offset 30 minutes (1800 seconds) from GMT year-round, with abbreviation `INT`, call:

```
> old.list <- timeZoneList(INT = timeZoneS(offset=1800))
```

The following command creates a time zone for another Hypothetical Country whose time is normally offset one hour from GMT. The country had (one hour changing) daylight savings time that started on the last Sunday in April and ended on the last Sunday in September through 1989, and then on the first Sundays in May and October thereafter. We abbreviate this time zone as `HCT`.

```
> old.list <- timeZoneList(HCT = timeZoneS(offset = 3600,
+ yearfrom = c(-1, 1990), yearto = c(1989, -1),
+ hasdaylight = c(T, T), dsextra = c(3600, 3600),
+ monthstart = c(4, 5), codestart = c(2, 3),
+ daystart = c(0, 0), xdaystart = c(0, 1),
+ timestart = c(7200, 7200), monthend = c(9, 10),
+ codeend = c(2, 3), dayend = c(0, 0), xdayend = c(0, 1),
+ timeend = c(7200, 7200)))
```

See the documentation for the `timeZoneS` function and the `timeZone` class for more information.

Daylight Savings and Built-In Time Zones

With the exception of GMT, the time zones built into Spotfire S+ automatically correct for summer time, also known as *daylight savings time*. Thus, for example, the default US Pacific time zones PST (Pacific Standard Time) and PDT (Pacific Daylight Time) refer to GMT minus 8 hours in the winter and GMT minus 7 hours in the summer, respectively. Despite the time difference, Spotfire S+ treats the two time zones identically.

A time that occurs within the one-hour time change period and which includes a summer time zone is poorly-defined: an hour is skipped in the spring and an hour is repeated in the autumn. Spotfire S+ accepts such times without warning, interpreting “illegal” times as summer time during the spring changeover and winter time during the autumn change. If your data include such times, you should use a time zone such as GMT that does not correct for summer time. Alternatively, you can redefine the summer and winter time zones to be fixed offsets from GMT. For example:

```
> timeZoneList(PDT = timeZoneS(offset = -25200),    # GMT-7
+     PST = timeZoneS(offset = -28800))             # GMT-8
```

If you choose to redefine time zones in this way, you should ensure that all times are correctly assigned to the summer and winter time zones as appropriate; Spotfire S+ will no longer make automatic corrections.

Creating and Displaying Time/Date Objects With Time Zones

Once the desired time zone names are defined as the names of the time zone list (see previous sections), you can create time/date objects with time zones by using one of the names as the `zone` argument in the `timeDate` or `timeCalendar` function. You can see the time zone by accessing the `time.zone` slot of the time/date object.

For example:

```
> x <- timeDate("Jan 1, 1998 2:04 PM", zone = "PST")
> x

[1] 01/01/1998 14:04:00.000

> x@time.zone

[1] "PST"

> x <- timeCalendar(m=1, d=1:10, y=1998, zone="PST")
```

Character strings containing times and dates with time zones can be converted to time/date objects using the `timeDate` function, in the same way as character strings without time zones. Since each element of a character vector potentially has a different time zone, if you do not supply a `zone` argument, the result will be converted to `GMT`:

```
> x <- timeDate(c("Jan 1, 1998 2:04 PM (PST)",
+ "July 1, 1998 10:54 AM (PDT)"))

> x

[1] 01/01/1998 22:04:00.000 07/01/1998 17:54:00.000

> x@time.zone

[1] "GMT"
```

By supplying a `zone` argument, you can display the result in the desired time zone:

```
> x <- timeDate(c("Jan 1, 1998 2:04 PM (PST)",
+ "July 1, 1998 10:54 AM (PDT)"), zone = "PST/PDT")

> x

[1] 01/01/1998 14:04:00.000 07/01/1998 10:54:00.000
```

You may also wish to change the output format of the time/date object, using the `format` argument, so that the time zone is displayed. The `%z` format specification inserts the time zone name. If the name contains a “/” character, it uses the part before the first “/” as the name for standard time and the part after for daylight time. The command below illustrates this.

```
> timeDate(c("Jan 1, 1998 2:04 PM (PST)",
+ "July 1, 1998 10:54 AM (PDT)"), zone = "PST/PDT",
+ format = "%02m/%02d/%Y %02H:%02M (%z)")

[1] 01/01/1998 14:04 (PST) 07/01/1998 10:54 (PDT)
```

Changing and Converting Between Time Zones

Internally, as mentioned previously, all times in Spotfire S+ are stored in GMT, and the time zone slot of a time/date object is an overlay for display and other calculations. Because of this storage architecture, time zone conversions in Spotfire S+ only require changing the character string in the time zone slot of your time/date object to a new valid time zone name. For example:

```
> x <- timeDate(c("Jan 1, 1998 2:04 PM (PST)",
+ "July 1, 1998 10:54 AM (PDT)"), zone = "PST/PDT",
+ format = "%02m/%02d/%Y %02H:%02M (%z)")
> x

[1] 01/01/1998 14:04 (PST) 07/01/1998 10:54 (PDT)

> x@time.zone <- "EST/EDT"
> x

[1] 01/01/1998 17:04 (EST) 07/01/1998 13:54 (EDT)
```

Spotfire S+ also supports another kind of time zone conversion that *does* change the internal time storage. This is useful if you have created a time/date object in the default `GMT` time zone that really should have the same displayed time of day but in a different time zone. For example, if your time/date object has a time of 2 PM GMT and you change the time zone slot to `EST`, the time would become 9 AM EST, but you really want it to be 2 PM EST. This situation typically arises when you create a time/date object from data that actually refers to a time zone other than `GMT` without using the `zone` argument or other time zone indicators in character data, when using the holiday generating functions, or when you have been using time/date objects without considering time zones and then realize they are important to your analysis. To make this type of time zone conversion, use the `timeZoneConvert` function:

```
> x <- timeDate(c("Jan 1, 1998 2:04 PM",
+ "July 1, 1998 10:54 AM"))
> x

[1] 01/01/1998 14:04:00.000 07/01/1998 10:54:00.000
```

```
> x@time.zone
[1] "GMT"

> x <- timeZoneConvert(x, "PST/PDT")
> x
[1] 01/01/1998 14:04:00.000 07/01/1998 10:54:00.000

> x@time.zone
[1] "PST/PDT"
```

Mathematical Operations and Alignment With Time Zones

Every time/date object (except those whose time zone is `GMT`) has two different times associated with each element of the vector: the time in the local time zone and the absolute GMT time. Most mathematical operations operate on the GMT time. For instance, if you compare two time/date objects using the standard Spotfire S+ comparison operators, their GMT times will be compared. If you perform defined arithmetic operations on time/date objects, these operations will also operate on the GMT time, but if the result is still a time/date object, it will not lose its time zone. For the most part, combining numbers and time interval objects with time/date objects will look as though it is done in the local time zone. The only exception is if the operation crosses a daylight savings time change, where if you add, for example, 3 hours, you might see an apparent 2-hour or 4-hour local zone time change. In addition, coercion using the `as` function to and from numeric objects is always done in GMT.

There are several operations that do consider the local time zone; display operations and conversion to and from character data are the main examples. In addition, the special functions that extract portions of the time/date object, such as `days`, `months`, `hours`, and `mdy`, extract them in the local time zone. It is also possible to align and merge time series objects using each object's local time zone (for example, treating 2 PM Pacific Standard Time, 2 PM GMT, and 2 PM Japan Standard Time as the same absolute time, instead of converting them all to GMT). This is done by setting the `localzone` argument to the `align` and `seriesMerge` functions to `TRUE`. For more information about aligning and merging, see section Interpolation and Alignment of Series and section Merging Series in Chapter 5, Time Series and Signal Basics.

TIME INTERVALS IN SPOTFIRE S+

Time intervals are another type of time-related data in Spotfire S+. Because they are not associated with a particular calendar date or time of day, they are different from the time/date objects we have been discussing so far in this chapter. There are two types of time interval objects in S-PLUS. The first is represented by the `timeSpan` class, which stores time intervals as an absolute number of days and milliseconds; when you subtract two `timeDate` objects in S-PLUS, you get a `timeSpan` object. The second type of time interval is represented by the `timeRelative` class, which stores time intervals in terms of less absolute units, such as business days, partial months, and years. The following sections show how to use these two types of time interval objects in S-PLUS, including how to create them from character data, display them, create time span objects from numeric data, perform basic subsetting and manipulation, and combine them with time/date objects.

Creating Time Span Objects from Character Data

One way to create a time span object is to subtract two time/date objects:

```
> timeDate(c("Jan 1 1998 3:05PM", "Jan 5 1998 4:10PM")) -
+ timeDate("Jan 1 1998 1AM")

[1] 0d 14h 5m 0s 0MS 4d 15h 10m 0s 0MS
```

In some cases, you may also want to create a time span object by converting a character string. This can be done using the `as` function or the `timeSpan` function:

```
> as(c("34 days 3 hours 1 min", "-18d -2h -5s"),
+ "timeSpan")

[1] 34d 3h 1m 0s 0MS -18d -2h 0m -5s 0MS

> timeSpan(c("34 days 3 hours 1 min", "-18d -2h -5s"))

[1] 34d 3h 1m 0s 0MS -18d -2h 0m -5s 0MS
```

As you can see, `timeSpan(x)` and `as(x,"timeSpan")` produce the same output when `x` is a character vector. The `timeSpan` function is more flexible than the `as` function because it also allows you to specify a different input format using the `in.format` argument:

```
> timeSpan(c("3:04:02", "5:23:45"), in.format = "%H:%M:%S")
```

```
[1] 0d 3h 4m 2s 0MS  0d 5h 23m 45s 0MS
```

Input formats are single-element character vectors consisting of input fields (which start with "%" and end with a letter) and other characters (such as ":", ".", "/") that must be matched exactly. Commonly used input specifications include:

`%d`	Input number of days as integer
`%y`	Input number of years (always 365 days) as integer
`%H`	Input number of hours as integer
`%M`	Input number of minutes as integer
`%S`	Input number of seconds as integer
`%N`	Input number of milliseconds as integer
`%#c`	Skip # characters
`%$c`	Skip rest of input string
`%w`	Skip one word

Notes

The default input format is stored in `options("tspan.in.format")`. There is also a list of sample formats in `format.timeSpan`.

Input specifications can be duplicated, since each one encountered simply adds the given time quantity to the time span being read.

See the help document on the `timeSpan` class for a complete description of time span input formats.

Displaying Time Span Objects

Time span objects have a default display and printing format, as seen in the previous section, that displays the number of days, hours, minutes, seconds, and milliseconds in the time span. You can also change the display format of a `timeSpan` object, for instance, to display weeks or years instead of a large number of days or to change the order of output. One way to do this is to include a `format` argument in the `timeSpan` function when you create the time span object:

```
> timeSpan(c("3:04:02", "5:23:45"), in.format = "%H:%M:%S",
+ format = "%02H:%02M:%02S")

[1] 03:04:02 05:23:45
```

You can also change the format on an existing time span object by accessing the `format` slot:

```
> x <- timeSpan(c("3:04:02", "5:23:45"),
+ in.format = "%H:%M:%S")
> x@format="%H hours %M minutes %S seconds"
> x

[1] 3 hours 4 minutes 2 seconds
[2] 5 hours 23 minutes 45 seconds
```

Output formats are single-element character vectors consisting of output fields (which start with "%" and end with a letter) and other characters (such as letters, ":", ".", "/") that are simply printed. Commonly used output specifications include:

`%d`	Print total number of days as integer
`%D`	Print number of days subtracting off full 365-day years as integer
`%E`	Print number of days subtracting off full 7-day weeks as integer
`%W`	Print number of full weeks as integer
`%y`	Print number of full 365-day years as integer
`%H`	Print number of hours subtracting off full days as integer
`%M`	Print number of minutes subtracting off full hours as integer

`%S`	Print number of seconds subtracting off full minutes as integer
`%s`	Print number of seconds subtracting off full days as integer
`%N`	Print number of milliseconds subtracting off full seconds as integer

Notes

To make an integer field print with a fixed width (padded with zeros if necessary), put `0#`, where # is your desired width between the `%` and the output code. For example, if the number of days is 7 and it is printed with `%d`, it will appear as "7"; if it is printed with `%02d`, it will appear as "07". If you prefer spaces instead of 0s, use, for example, `%2d`. Be sure to leave room for a "-" sign if it is possible that the number could be negative.

The default input format is stored in `options("tspan.out.format")`. There is also a list of sample formats in `format.timeSpan`.

See the help document on the `timeSpan` class for a complete description of time span output formats.

Creating Time Span Objects from Numeric Data

There are two ways to convert numeric data into time span objects. First, you can use the `as` function to convert numbers representing days and fractions of days directly to time spans:

```
> as(c(-1.2, 3.4, 5.6), "timeSpan")

[1] -1d -4h -48m 0s 0MS 3d 9h 36m 0s 0MS
[3] 5d 14h 24m 0s 0MS
```

Second, you can use the `timeSpan` function's `julian` and `ms` arguments to convert integer vectors of days and milliseconds into time spans:

```
> timeSpan(julian = 1:4, ms = (5:8) * 1000)

[1] 1d 0h 0m 5s 0MS 2d 0h 0m 6s 0MS
[3] 3d 0h 0m 7s 0MS 4d 0h 0m 8s 0MS
```

Basic Operations on Time Span Objects

Time span objects can, in most cases, be manipulated like most other S-PLUS objects. For instance, subscripting works the same as for other vectors, and you can use the `length` function to count or set the number of elements. You can concatenate time span objects using the `concat` function, and you can convert them to character, numeric, and integer vectors using the `as` function.

Various mathematical operations are also defined for time span objects. You can perform addition, subtraction, multiplication, and division between time spans and numbers, addition and subtraction between time spans, and addition and subtraction between time spans and time/date objects. Time span objects can also be compared using the usual comparison operators. Many other basic S-PLUS vector manipulation functions also work on time span objects; `mean`, `quantile`, `max`, `floor`, `sort`, `rev`, `match`, `unique`, and `cut` are some examples. Operations that do not make sense for time spans cannot be performed (for example, taking the logarithm or the cosine).

A few examples:

```
> x <- as(c(-1.2, 3.4, 5.6), "timeSpan")
> y <- timeSpan(julian = 1:4, ms = (5:8) * 1000)
> x[1:2] - y[3:4]

[1] -4d -4h -48m -7s 0MS 0d -14h -24m -8s 0MS

> x * 5

[1]  -6  17  28

> floor(x)

[1] -2d 0h 0m 0s 0MS 3d 0h 0m 0s 0MS 5d 0h 0m 0s 0MS
```

Notice that the operation `x*5` returns a numeric vector, while `floor(x)` returns an object of class `"timeSpan"`. When working with `timeSpan` objects, basic arithmetic operations such as multiplication return numeric objects. This allows you to pass the results into other functions that may not have `timeSeries` methods, without requiring you to perform additional conversions.

Relative Time Objects

Unlike time span objects, which store absolute time differences (for example, numbers of milliseconds), relative time objects store time intervals in units such as weekdays, months, and business days, whose absolute number of days and milliseconds depends on what time/date objects they are combined with. Relative time is stored in Spotfire S+ in the `timeRelative` class, which stores a vector of character strings encoding the time intervals and a (possibly empty) vector of holidays for business day calculations. Each element of the encoding vector is a character string consisting of space-separated fields in the form `"{+-}[a]#abb"` (for example, `"+a3day"`, `"-5hr"`, `"+a0mth"`) containing a required sign, either `"+"` or `"-"`, followed by an optional `"a"` that, if present, means to align the result; see the table below for an exact meaning. The optional `"a"` is followed by a positive integer, represented here by `"#"`. It is also possible to specify 0 for the integer if aligning; see the table below. The integer is followed by one of the relative time field abbreviations from the following table:

`ms`	Add/subtract milliseconds; *a* aligns to nearest # milliseconds within the second, where # must be a divisor of 1000 and less than 1000 (for example, 500 aligns to seconds or 1/2 seconds); 0 is not allowed.
`sec`	Add/subtract seconds; *a* aligns to nearest # seconds within the minute, where # must be a divisor of 60 and less than 60 (for example, 15 aligns to 0, 15, 30, or 45 seconds past the minute); 0 goes to the beginning of the current second, independent of sign.
`min`	Add/subtract minutes; *a* aligns to nearest # minutes within the hour, where # must be a divisor of 60 and less than 60 (for example, 15 aligns to 0, 15, 30, or 45 minutes after the hour); 0 goes to the beginning of the current minute, independent of sign.
`hr`	Add/subtract hours; *a* aligns to nearest # hours within the day, where # must be a divisor of 24 and less than 24 (for example, 6 aligns to midnight, 6AM, noon, or 6PM); 0 goes to the beginning of the current hour, independent of sign.
`day`	Add/subtract days; *a* aligns to nearest # days within the month, starting with the first, where # must be a less than the number of days in the month (for example, 2 aligns to the 1st, 3rd, 5th, etc., with the time midnight); 0 goes to the beginning of the current day, independent of sign.

`wkd`	Add/subtract weekdays; *a* causes the first added or subtracted weekday to possibly be a fraction of a day to move to the next or previous midnight on a weekday morning, and then whole additional days are added or subtracted to make up # weekdays; 0 goes to the beginning of the day, or the closest weekday before if it is not a weekday, independent of sign.
`biz`	Add/subtract business days (weekdays that are not holidays); *a* causes the first added or subtracted business day to possibly be a fraction of a day to move the next or previous midnight on a business day morning, and then whole additional days are added or subtracted to make up # business days; 0 goes to the beginning of the day, or the closest business day before if it is not a business day, independent of sign.
`sun` `mon` `tue` `wed` `thu` `fri` `sat`	Add/subtract Sundays, Mondays, Tuesdays, Wednesdays, Thursdays, Fridays, or Saturdays; *a* causes the first added or subtracted day to possibly be a fraction of a day or week to move the next or previous midnight on the selected day's morning, and then whole additional weeks are added or subtracted to make up # of the selected day; 0 goes to the beginning of the day, or the closest selected day before if it is not the right day, independent of sign.
`wk`	Add/subtract weeks; *a* is not allowed. Use the specific weekday or just `"day"`, `"wkd"` or `"biz"` to align.
`tdy`	Add/subtract ten-day periods of months (ten-day periods begin on the first, 11th, and 21st of the month but not the 31st). Without *a*, the day number of the result will be 1, 11, or 21, adding # partial or entire ten-day periods to get there. If *a* is used, # must be either 1 (any), 2 (11th only), or 3 (21st only), and the time will be midnight; 0 goes to the beginning of the current ten-day period, independent of sign.
`mth`	Add/subtract months; *a* aligns to nearest # months within the year, starting with January, and # must be a divisor of 12 and less than 12 (for example, 3 aligns to January 1, April 1, July 1, October 1 at midnight); 0 goes to the beginning of the current month, independent of sign.

`qtr`	Add/subtract quarters; *a* aligns to nearest # quarters within the year, and # must be either 1 or 2 (for example, 2 aligns to Jan 1 or Jul 1 at midnight); 0 goes to the beginning of the current quarter, independent of sign.
`yr`	Add/subtract years; *a* aligns to nearest # years (for example, 5 aligns to Jan 1 at midnight in 1995, 2000, 2005, etc.); 0 goes to the beginning of the current year, independent of sign.

For example, to create a relative time object that you could add to a time/date object (or vector of them) to take each element to the third Friday of the month, you would first get to the beginning of the month by aligning, and then subtract 1 Friday to go to the last Friday of the previous month, and then add 3 Fridays:

```
> rt <- timeRelative("-a0mth -1fri +3fri")
> timeDate(c("1/5/1998", "2/26/1998"),
+ format = "%a %m/%d/%Y") + rt

[1] Fri 1/16/1998 Fri 2/20/1998
```

As you can see, relative time objects are created by the `timeRelative` function; for business day relative times, it also takes a `holiday` argument giving a vector of dates that are to be considered holidays when combining the relative time with a time/date object. You can also create relative time objects by coercing character data directly (using the `as` function), but you will not be able to provide holidays.

Once they are created, relative time objects can be subscripted like other S-PLUS objects, and they can be added to and subtracted from time/date objects. If either the relative time or time/date object is shorter than the other, the shorter one will be reused cyclically in the normal Spotfire S+ manner. Relative time objects can also be added to or subtracted from each other or multiplied by integers. Since combination of relative time and time/date objects goes from left to right in the string, adding two relative times together is the same as pasting their strings together. It should also be noted that because relative time objects do not represent absolute quantities of time, generally if `x` and `y` are two relative time objects, `x+y` is not the same as `y+x`.

TIME SEQUENCES IN SPOTFIRE S+

There are two ways to create and store regular sequences of times and dates in Spotfire S+. First, the `timeSeq` function can be used to create a regular sequence and store it in a time/date object. This is similar to the `seq` function, which creates sequences of numbers, and it defaults to a one day increment:

```
> timeSeq(from="1/1/1992", to="1/10/1992")

 [1] 01/01/1992 01/02/1992 01/03/1992 01/04/1992
 [5] 01/05/1992 01/06/1992 01/07/1992 01/08/1992
 [9] 01/09/1992 01/10/1992

> timeSeq(from = "1/6/1992", to = "12/6/1992",
+ by = "months")

 [1] 01/06/1992 02/06/1992 03/06/1992 04/06/1992
 [5] 05/06/1992 06/06/1992 07/06/1992 08/06/1992
 [9] 09/06/1992 10/06/1992 11/06/1992 12/06/1992
```

As you can see, the default behavior of the `timeSeq` function is to add the `by` argument's time units to the start of the sequence, until the end is reached. The `align.by` argument allows you to create a sequence that is aligned to the `by` units, in this case always landing on the beginning of months. The `extend` argument is used to control whether the sequence extends outward (`T`) or inward (`F`, the default) from the `from` and `to` arguments if they are not aligned to the `by` units:

```
> timeSeq("1/3/1992", "12/5/1992", by = "months",
+ align.by = T)

 [1] 02/01/1992 03/01/1992 04/01/1992 05/01/1992
 [5] 06/01/1992 07/01/1992 08/01/1992 09/01/1992
 [9] 10/01/1992 11/01/1992 12/01/1992

> timeSeq("1/3/1992", "12/5/1992", by = "months",
+ align.by = T, extend = T)

 [1] 01/01/1992 02/01/1992 03/01/1992 04/01/1992
 [5] 05/01/1992 06/01/1992 07/01/1992 08/01/1992
 [9] 09/01/1992 10/01/1992 11/01/1992 12/01/1992
[13] 01/01/1993
```

The `timeSeq` function also allows you to use a relative time or time span object as the `by` argument, which is especially useful when the desired increment is not a whole number of days, months, etc.

In addition, you can use the `timeSeq` function to generate sequences with exceptions and additions. Exceptions are periods when the generated times and dates should be dropped, represented by an event object; event objects are described in section Representing Events in Spotfire S+. Additions are times and dates that should be added to the sequence. For instance, to generate a monthly sequence on the 25th of each month in the last quarter of 1998 and the first quarter of 1999, with the exception of December, when you want to use the 24th:

```
> timeSeq("10/25/1998", "3/25/1999", by = "months",
+ exceptions = timeDate("12/25/1998"),
+ additions = timeDate("12/24/1998"))

[1] 10/25/1998 11/25/1998 12/24/1998 01/25/1999
[5] 02/25/1999 03/25/1999
```

Since the additions and the starting/ending times of the exceptions can also be time sequences, the `timeSeq` function can be used to generate very complicated sequences.

Besides generating time/date objects in regular or semi-regular sequences using the `timeSeq` function, you can also store regular time sequences in a compact object of class `"timeSequence"`. These objects are generated with the `timeSequence` function, which takes exactly the same arguments as `timeSeq`. Instead of returning a vector containing every time and date in the sequence, however, it returns an object that stores only the starting and ending times, length and/or increment (as a time interval object), exceptions, and additions for the sequence. Subsequently, a time and date vector is generated from a `timeSequence` object whenever it is needed for a calculation; for efficiency you may find it better to convert to class `"timeDate"` using the `as` function for repeated calculations. Here is what the previous sequence looks like as a time sequence object:

```
> x <- timeSequence("10/25/1998", "3/25/1999",
+ by = "months", exceptions = timeDate("12/25/1998"),
+ additions = timeDate("12/24/1998"))

> x
```

```
from:   10/25/1998
to:     03/25/1999
by:     +1mth
exceptions: 1
additions: 1
[1] 10/25/1998 11/25/1998 12/24/1998 ...       03/25/1999
```

```
> as(x, "timeDate")
```

```
[1] 10/25/1998 11/25/1998 12/24/1998 01/25/1999
[5] 02/25/1999 03/25/1999
```

NUMERIC SEQUENCES IN SPOTFIRE S+

There are two ways to create and store regular sequences of numbers in Spotfire S+. First, the `seq` function or its compact `":"` operator form can be used to create a regular sequence of numbers and store it in a numeric vector:

```
> seq(from=1, to=10, by=1)

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

For storage efficiency, regular sequences of numbers can also be stored in an object of class `"numericSequence"`, which stores the information needed to reconstruct the sequence instead of storing all of the numbers in the sequence. You can create a numeric sequence object with the `numericSequence` function:

```
> numericSequence(from=1, to=10, by=1)

from:   1
to:     10
by:     1
[1]  1   2   3  ... 10

> numericSequence(to=10, by=1, length=10)

to:     10
by:     1
length: 10
[1]  1   2   3  ... 10
```

You can convert a numeric sequence object into a numeric vector using the `as` function. Some arithmetic operations can also be performed directly on numeric sequence objects, with automatic coercion to numeric vectors. However, you may find it more efficient to convert to numeric for long sequences of calculations. Some examples:

```
> x <- numericSequence(from=1, to=10, by=1)
> as(x, "numeric")

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

> x + c(1:5, 5:1)
```

```
[1]  2  4  6  8 10 11 11 11 11 11

> x == 3

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

You can also use the `length` function to calculate the length of a numeric sequence object; if you subscript a numeric sequence object, you will end up with an ordinary numeric vector. However, subscript and length replacement operations must be done by first coercing the object to an ordinary numeric vector.

REPRESENTING EVENTS IN SPOTFIRE S+

There are several types of data, broadly classified as *events*, that can play an important role in data analysis, especially in the analysis of financial data. For example, the first Gulf War was an one-time event that had large effects on the oil markets, so if you were studying oil prices, you might want to analyze the effects of that war, and if so you would need to keep track of its starting and ending dates. Continuing with the oil price example, another analysis might consider the effects of OPEC meetings, for which you would need to keep track of the starting and ending times of each meeting. A shorter-term analysis of the price volatility of oil futures might take into account the opening and closing times of the futures exchange each day and the holidays and weekends when the exchange is closed.

In each of these examples, a single or recurring "event" is associated with a starting and ending time and date for each occurrence, along with some identifying information in some cases. To store and represent such data, Spotfire S+ has the `timeEvent` class, which stores a starting time, ending time, and optional ID string for each occurrence of the event; the starting and ending times/dates can be either a time/date or time sequence object. To create an event object, use the `timeEvent` function:

```
> timeEvent(start = holiday.Christmas(1995:2000),
+ ID = 1995:2000)

  ID                   start                     end
1995 12/25/1995 00:00:00.000 12/25/1995 23:59:59.999
1996 12/25/1996 00:00:00.000 12/25/1996 23:59:59.999
1997 12/25/1997 00:00:00.000 12/25/1997 23:59:59.999
1998 12/25/1998 00:00:00.000 12/25/1998 23:59:59.999
1999 12/25/1999 00:00:00.000 12/25/1999 23:59:59.999
2000 12/25/2000 00:00:00.000 12/25/2000 23:59:59.999
```

```
> timeEvent(start = timeCalendar(m=5, d=4:8, y=1998, h=8),
+ end = timeCalendar(m=5, d=4:8, y=1998, h=17),
+ ID = day.name[2:6])
```

```
       ID                   start                     end
Monday    05/04/1998 08:00:00.000 05/04/1998 17:00:00.000
Tuesday   05/05/1998 08:00:00.000 05/05/1998 17:00:00.000
Wednesday 05/06/1998 08:00:00.000 05/06/1998 17:00:00.000
Thursday  05/07/1998 08:00:00.000 05/07/1998 17:00:00.000
Friday    05/08/1998 08:00:00.000 05/08/1998 17:00:00.000
```

Event objects can be subscripted in the normal way, and they support a few standard S-PLUS functions such as `match`, `is.na`, `duplicated`, and `concat`, but they cannot be used directly in mathematical operations. You can use the `length` function to calculate or change the length of an event object. One of their primary uses is to specify exceptions in time sequence generation; see the section Time Sequences in Spotfire S+ for more details.

WRITING FUNCTIONS IN Spotfire S+ 7

INTRODUCTION

Programming in Spotfire S+ consists largely of writing functions. The simplest functions arise naturally as shorthand for frequently-used combinations of S-PLUS expressions.

For example, consider the *interquartile range*, or IQR, of a data set. Given a collection of data points, the IQR is the difference between the upper and lower (or third and first) quartiles of the data. Although Spotfire S+ has no built-in function for calculating the IQR, it does have functions for computing quantiles and differences of numeric vectors. The following two commands define and test a function that returns the IQR of a numeric vector.

```
> iqr <- function(x) { diff(quantile(x, c(0.25, 0.75))) }
> iqr(lottery.payoff)

   75%
169.75
```

You can build more complicated functions either by adding new features incrementally to simpler functions, or by designing whole programs from scratch. As your functions grow more complex, proper use of programming features becomes more important.

This chapter describes the basic techniques for writing functions in Spotfire S+. It first outlines the structure underlying all S-PLUS functions, and then describes some of the most useful functions for manipulating data. A section on organizing computations gives tips on designing functions that take advantage of the strengths of Spotfire S+. Later sections introduce techniques for argument handling, error handling, input and output, and wrap-up actions. From these few simple tools and techniques, you can build many useful functions.

Windows Users

To run the examples in this chapter, you will need to create functions with an editor. There are many different approaches to editing functions in Spotfire S+, but the simplest way to get started is with the `Edit` function. The built-in function `Edit` creates a function template with the proper structure when called with a name that does not correspond to an existing S-PLUS object. Thus, to create a new function called `newfunc`, call `Edit` as follows:

```
> Edit(newfunc)
```

Edit the template as desired in the **Script** window that appears. To source in the modified function, select **Script** ▶ **Run** from the menu, press the F10 key, or use the **Run** button on the **Script** toolbar.

To edit an existing function, call `Edit` using the function's name. Alternatively, right-click on the function's name in the **Object Explorer** and select **Edit** from the context-sensitive menu. Refer to the section Editing Objects (page 14) for more details.

UNIX Users

To run the examples in this chapter, you will need to create functions with an editor. There are many different approaches to editing functions in Spotfire S+, but the simplest way to get started is with the `fix` function. You can use `fix` to either create a new function or modify an existing one, using an external editor such as **vi**. Refer to the section Editing Objects (page 14) for more details.

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*. In this section, we discuss these components in detail. In addition, we discuss programming concepts such as return values, side effects, and coercion. For completeness, we also include sections on elementary functions, complex operations, and logical operators.

Function Names and Operators

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

Operators are special functions for performing mathematical or logical operations on one or two arguments. They are most convenient to use in *infix* form, in which they appear between two arguments. Familiar examples of operators are `+`, `-`, and `*`. The names of such functions consist of the symbol used to represent them enclosed by double quotes. Thus, `"+"` is the function name corresponding to the addition operator `+`. You can use names to call operators as functions in the ordinary way. For example, the call `"+"(2,3)` is represented by `2+3` in infix form; both commands return the number 5.

A complete list of built-in operators is provided in Table 7.1. In addition to the predefined operators in the table, Spotfire S+ allows you to write your own infix operators. For more details, see the section Operators (page 241).

Operators listed higher in Table 7.1 have higher precedence than those listed below. Operators on the same line in the table have equal precedence, and evaluation proceeds from left to right when more than one of these operators appear in an expression. For example, consider the command:

```
> 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 in the expression. Therefore, it is performed first: `64/19=3.368421`. Next comes the multiplication: `3.368421*2=6.736842`. Finally, Spotfire S+ performs the addition and subtraction: `7+5-6.736842=5.263158`.

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

```
> (7 + 5 - 8^2) / (19 * 2)
[1] -1.368421
```

The integer divide operator in S-PLUS, `%/%`, produces an integral quotient. For two numbers a and b, the S-PLUS expression `a%/%b` computes q in Euclid's algorithm: $a = qb + r$ where $0 \le r < b$. The modulus operator `%%` computes the remainder r.

Table 7.1: *Precedence of operators. Operators listed higher in the table have higher precedence than those listed below, and operators on the same line have equal precedence.*

Operator	**Use**
`$`	component selection
`@`	slot selection
`[  [[`	subscripts, elements
`^`	exponentiation
`-`	unary minus
`:`	sequence operator
`%%  %/%  %*%`	modulus, integer divide, matrix multiply
`*  /`	multiply, divide
`+  -  ?`	add, subtract, help

Table 7.1: *Precedence of operators. Operators listed higher in the table have higher precedence than those listed below, and operators on the same line have equal precedence.*

Operator	**Use**
`<` `>` `<=` `>=` `==` `!=`	comparison
`!`	not
`&` `|` `&&` `||`	logical and, logical or
`~`	formulas
`<<-`	permanent assignment
`<-` `->` `_` `=`	assignments

Note

When using the `^` operator, the exponent must be an integer if the base is a negative number. If you require a complex result when the base is negative, be sure to coerce it to mode `"complex"`. See the section Operations on Complex Numbers (page 162) for more details.

Another special type of function is the *replacement* or *left-side* function. It has the appearance of an ordinary function on the left side of an assignment. For example, the expression `dim(x) <- c(3,4)` uses the replacement function `"dim<-"`. Spotfire S+ 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`.

Replacement functions can be defined for *extraction functions*, which are functions designed to return some specific portion or attribute of a data object. Common extraction functions are the subscript operator `[]`, the `dim` function, and the `names` function. For details, see the online help files for these functions and the section Extraction and Replacement Functions (page 242).

Arguments

Arguments to a function specify the data to be operated on, and also pass processing parameters to the function. Not all functions accept arguments. For example, the `date` function can only be called with the syntax `date()`:

```
> args(date)
function()
```

In contrast, the `lm` function accepts many arguments:

```
> args(lm)
function(formula, data, weights, subset, na.action,
  method = "qr", model = F, x = F, y = F, contrasts = NULL,
  ...)
```

Functions without arguments are, by design, rigid and single-purpose. Their behavior can be modified only by editing the function. Arguments allow you to build multi-purpose functions with behavior that can be easily modified whenever a function is called. For a complete discussion of allowable argument lists, see the section Specifying Argument Lists (page 199)

The Function Body

The *body* of a function is the part that actually does the work. It consists of a sequence of S-PLUS statements and 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; we recommend including them in all of your functions because it makes maintenance less accident-prone. By adding braces when you define a single-line function, you ensure they won't be forgotten when you add functionality to it.

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

Return Values and Side Effects

Functions are designed to accomplish something, and if everything goes as planned, a function accomplishes something every time it is called. Most functions do one thing: return a value. A *return value* can be any valid S-PLUS expression, although it is usually a transformed version of the input data. In general, values returned from functions are not automatically saved. Therefore, most calls to functions also involve an assignment:

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

In this expression, the return value from the function `f` on the input `x` is preserved in the object `y` for further analysis.

Note

In compiled languages such as C and Fortran, you can pass arguments directly to a function that modifies the argument values in memory. In Spotfire S+ however, all arguments are passed *by value*. This means that only copies of the arguments are modified throughout the body of a function.

Sometimes, you may want a function to do something besides return a S-PLUS expression. For instance, you may want to print something, draw a graph, or change some Spotfire S+ session options. Because the main goal of functions is to return values, these other actions are collectively called *side effects*. The section Data Output (page 208) discusses return values and side effects in more detail.

The combination of a function's side effects and its return value can be used to good advantage in some situations. For example, the `options` function has the side effect of changing the current Spotfire S+ session options. It also returns a value that consists of the options in effect before the current call. Thus, you can use `options` within a function not only to change the options in effect, but also to save the old options for restoration when the function exits. The following commands illustrate this:

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

By assigning the return value of `options` to `options.old`, we save the old `width` setting. The side effect of the first command changes `options` to use a `width` of 55 characters; this takes place whether or not we assign the return value. The `on.exit` function performs a given set of actions when the calling function exits. In this example, `on.exit` restores the old `width` value at the end of the calling function.

Elementary Functions

In addition to the infix operators introduced in the section Function Names and Operators (page 153), Spotfire S+ includes a variety of elementary mathematical functions that act in a *vectorized* way on numeric data sets. That is, the functions manipulate numeric vectors the same way as single numeric elements. The elementary functions include the familiar trigonometric and exponential functions, as well as several functions for computing numerical results.

The functions listed in Table 7.2 are the vectorized math functions implemented internally as part of the S-PLUS language. Spotfire S+ has many other built-in mathematical functions, some of which are written wholly in the S-PLUS language and some of which are written to take advantage of existing algorithms in Fortran or C. See Chapter 36, Mathematical Computing in Spotfire S+, in the *Guide to Statistics, Volume 2* for more information.

Table 7.2: *Common elementary mathematical functions.*

Name	Operation
`sort, rev`	the input sorted in ascending or reverse order
`sqrt`	square root
`abs`	absolute value
`sin, cos, tan`	trigonometric functions
`asin, acos, atan`	inverse trigonometric functions
`sinh, cosh, tanh`	hyperbolic trigonometric functions
`asinh, acosh, atanh`	inverse hyperbolic trigonometric functions
`exp, log`	exponential and natural logarithm (base *e*)
`log10`	common logarithm (base 10)
`logb`	logarithm for bases other than *e* and 10

Table 7.2: *Common elementary mathematical functions. (Continued)*

Name	Operation
`gamma, lgamma`	gamma function and its natural logarithm
`ceiling`	closest integer not less than the input
`floor`	closest integer not greater than the input
`trunc`	closest integer between the input and zero
`round`	closest integer to the input
`signif`	the input rounded to a specified number of significant digits
`cummax, cummin`	cumulative maximum and minimum
`cumsum, cumprod`	cumulative sum and product
`pmax, pmin`	parallel maximum and minimum

Examples

Each function in Table 7.2 acts element-by-element on its argument. For example:

```
> M <- matrix(c(12,2,19,15,9,14,6,2,11,10,7,19), nrow=3)
> M

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

> sqrt(M)

         [,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(M)

           [,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
```

Note that both `sqrt(M)` and `tan(M)` return objects that are the same shape as `M`. The element in the *i*th row and *j*th column of the matrix returned by `sqrt(M)` is the square root of the corresponding element in `M`. Likewise, the element in the *i*th row and the *j*th column of `tan(M)` is the tangent of the corresponding element (assumed to be in radians).

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

```
> y <- c(-2.6, 1.5, 9.7, -1.0, 25.7, -4.6, -7.5, -2.7, -0.6,
+ -0.3, 2.8, 2.8)
> 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 accepts an optional argument `digits` that allows you to specify how many digits to include after the decimal point:

```
> round(sqrt(M), 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
```

The section Formatting Output (page 209) provides examples that further illustrate the `round` function.

Integer Arithmetic

By default, Spotfire S+ performs integer arithmetic if all arguments are integers, and real arithmetic if any arguments are real. In particular, if you pass an integer argument to a built-in function, Spotfire S+ attempts to return a integer value. If an integer value cannot be computed for the expression, Spotfire S+ returns `NA`. Earlier versions of S-PLUS automatically coerced integers to real numbers for storage purposes and performed real arithmetic by default. This changed in S-PLUS 5.x, however, and now the coercion must be done explicitly.

For example, here is the code for a S-PLUS function that computes the factorial of a number. We discuss this function in more detail in the section Wrap-Up Actions (page 237):

```
fac1024 <- function(n)
{
  old <- options(expressions = 1024)
  on.exit(options(old))
  if(n <= 1) { return(1) }
  else { n * Recall(n-1) }
}
```

If we call `fac1024` with `n=12` it works fine, but `n=13` causes it to return `NA`:

```
> fac1024(12)
[1] 479001600

> fac1024(13)
[1] NA
```

This is because Spotfire S+ attempts to compute an integer value for `13!` and overflows in the process. To force Spotfire S+ to compute real solutions, you must coerce the argument to a real number as follows:

```
> fac1024(13.0)
[1] 6227020800
```

Alternatively, we can replace the third line in the body of `fac1024` so that it always performs real arithmetic:

```
if(n <= 1) { return(1.0) }
```

With the function defined like this, the call `fac1024(13)` finishes without overflowing.

Operations on Complex Numbers

You represent complex literals in Spotfire S+ as a sum of the form $a + bi$, where a and b are real numbers. In general, arithmetic operations on complex numbers work as you would expect. Because the addition and subtraction operators have lower precedence than the `*`, `/`, and `^` operators, though, you must use parentheses to group complex arguments in most cases:

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

> (2+3i)^(3+2i)
[1] 4.714144-4.569828i
```

Warning

Do not leave any space between the real number b and the symbol `i` when defining complex numbers. If space is included between b and `i`, the following syntax error is returned:

```
Problem: Syntax error: illegal name ("i")
```

By default, Spotfire S+ 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 built-in function, Spotfire S+ attempts to return a real value. If a real value cannot be computed for the expression, Spotfire S+ returns `NA` and issues a domain error message. For example, here is the result when we pass the real number `-1` to the built-in square root function `sqrt`:

```
> sqrt(-1)
[1] NA
```

To force Spotfire S+ to consider complex solutions, you must coerce the arguments to mode `"complex"`, typically by using the function `as`:

```
> sqrt(as(-1, "complex"))
[1] 6.123032e-017+1i
```

Note that the real part of the result, `6.123032e-017`, is essentially equal to zero. Thus, (to machine precision) Spotfire S+ returns `1i` as the square root of -1, which is what we expect. Alternatively, you can include a zero-valued imaginary part to coerce real numbers to mode `"complex"`:

```
> sqrt(-1+0i)
```

```
[1] 6.123032e-017+1i
```

In addition to the ordinary operators and elementary mathematical functions, Spotfire S+ provides five special operators for manipulating complex numbers: `Re`, `Im`, `Mod`, `Arg`, and `Conj`. The `Re` and `Im` functions extract the real and imaginary parts, respectively, from a complex number. For example:

```
> x <- as(-3, "complex")
> x^(1/3)
[1] 0.7211248+1.249025i

> Re(x^(1/3))
[1] 0.7211248

> Im(x^(1/3))
[1] 1.249025
```

The `Conj` function returns the conjugate of a complex number:

```
> Conj(x^(1/3))
[1] 0.7211248-1.249025i
```

The `Mod` and `Arg` functions return the modulus and argument, respectively, for the polar representation of a complex number:

```
> Mod(2 + 2i)
[1] 2.828427

> Arg(2 + 2i)
[1] 0.7853982
```

Summary Functions

The mathematical operators and functions introduced so far act element-by-element, generally returning a value the same length and mode as the input data. Spotfire S+ also includes a number of functions for summarizing data. *Summary functions* accept an input vector or matrix and return a single value that summarizes the data in some way. For example, the `sum` and `prod` functions return the sum and product, respectively, of their arguments. Other useful summary functions are listed in Table 7.3. For details on any of the functions listed in the table, see the online help or Chapter 4, Descriptive Statistics, in the *Guide to Statistics, Volume 1*.

Table 7.3: *Common functions for summarizing data.*

Name	Operation
`min, max`	Return the smallest and largest values of the input arguments.
`range`	Returns a vector of length two containing the minimum and maximum of all the elements in all the input arguments.
`mean, median`	Return the arithmetic mean and median of the input arguments. The optional `trim` argument to `mean` allows you to discard a specified fraction of the largest and smallest values.
`var`	Returns the variance of a vector, the variance-covariance of a matrix, or covariances between matrices or vectors.
`stdev`	Returns the standard deviation of a numeric vector.

`quantile`	Returns user-requested sample quantiles for a given data set. For example,
	`> quantile(corn.rain, c(0.25, 0.75))`
	`   25%    75%`
	` 9.425 12.075`

`mad`	Returns the median absolute deviation of a numeric vector.
`cor`	Returns the correlation matrix of a data matrix, or correlations between matrices or vectors.
`skewness, kurtosis`	Return the skewness and kurtosis of a numeric vector.
`summary`	Returns the minimum, maximum, first and third quartiles, mean, and median of a numeric vector.

Comparison and Logical Operators

Table 7.4 lists the S-PLUS operators for comparison and logic. Comparisons and logical operations are frequently convenient for such tasks as extracting subsets of data. In addition, conditionals using logical comparisons play an important role in the flow of control in functions, as we discuss in the section Organizing Computations (page 185).

Table 7.4: *Logical and comparison operators.*

Operator	Explanation	Operator	Explanation
`==`	equal to	`!=`	not equal to
`>`	greater than	`<`	less than
`>=`	greater than or equal to	`<=`	less than or equal to
`&`	vectorized AND	`|`	vectorized OR
`&&`	control AND	`||`	control OR
`!`	not		

Notice that S-PLUS has two types of logical operators for AND and OR operations. Table 7.4 refers to the two types as "vectorized" and "control." The *vectorized operators* evaluate AND and OR expressions element-by-element, returning a logical vector containing `TRUE` and `FALSE` as appropriate. For example:

```
> x <- c(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
[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 F T T T T

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

In contrast, the *control operators* are used to construct conditional statements in `if` or `else` statements. The expressions in such statements are expected to have a single logical value, rather than a vector of logical values.

The control operators have the additional property that they are evaluated only as far as necessary to return a correct value. For example, consider the following expression for some numeric vector `y`:

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

The `any` function evaluates to `TRUE` if any of the elements in any of its arguments are true; it returns `FALSE` if all of the elements are false. Likewise, the `all` function evaluates to `TRUE` if all of the elements in all of its arguments are true; it returns `FALSE` if there are any false elements. Spotfire S+ initially evaluates only the first condition in the above expression, `any(x > 1)`. After determining that `x > 1` for some element in `x`, only then does Spotfire S+ proceed to evaluate the second condition, `all(y < 0)`.

Similarly, consider the following command:

```
> all(x >= 1) || 2 > 7
[1] T
```

Spotfire S+ stops evaluation with `all(x >= 1)` and returns `TRUE`, even though the statement `2 > 7` is false. Because the first condition is true, so is the entire expression.

Logical comparisons involving the symbolic constants `NA` and `NULL` always return `NA`, regardless of the type of operator used. For example:

```
> y <- c(3, NA, 4)
> y
[1]  3 NA  4

> y > 0
[1]  T NA  T

> all(y > 0)
[1] NA
```

To test whether a value is missing, use the function `is.na`:

```
> is.na(y)
[1] F T F
```

To test whether a component of a list or an attribute of an object is null, use the `is.null` function:

```
> is.null(names(kyphosis))
[1] F

> is.null(names(letters))
[1] T
```

For more details on functions such as `is.na` and `is.null`, see the section Testing and Coercing Data (page 169).

Assignments

As we have mentioned, data objects are created in Spotfire S+ by assigning values to names. We saw in the section Syntax of S-PLUS Expressions (page 7) that legal names consist of letters, numbers, and periods, and cannot begin with a number. The most common form of assignment in Spotfire S+ uses the *left assignment operator* `<-`, which may also be written as the equals sign = or a single underscore _ to save typing. The standard syntax is one of three forms:

- *`name <- expression`*
- *`name = expression`*
- *`name _ expression`*

Spotfire S+ interprets the expression on the right side of the assignment operator and returns a value. The value is then assigned to the name on the left side of the operator.

Because the underscore is a S-PLUS assignment operator, it is extremely important to remember that it cannot be used in function and object names, unlike in many other languages. In addition, it is deprecated as an assignment operator, so it may not be supported in future releases of Spotfire S+. See the section Syntax of S-PLUS Expressions (page 7) for more details.

Warning

In addition to object assignments, the equals sign is used for argument assignments within a function definition. Because of this, there are some ambiguities that you must be aware of when using the equals sign as an assignment operator. For example, the command

```
> print(x <- myfunc(y))
```

assigns the value from `myfunc(y)` to the object `x` and then prints `x`. Conversely, the command

```
> print(x = myfunc(y))
```

simply prints the value of `myfunc(y)` and does not perform an assignment. This is because the `print` function has an argument named `x`, and argument assignment takes precedence over object assignment with the equals sign. Because of these ambiguities, we discourage the use of the equals sign for left assignment.

Assignments made at the Spotfire S+ prompt are performed in the current working directory. Assignments within functions are local, and are performed in the *frame* in which the function is evaluated. This means that you can freely assign values to names within functions without overwriting existing objects that might share the same name. Frames are discussed in full in Chapter 2, Data Management.

Equivalent to the left assignment operator is *right assignment operator*, which appears in the form *`expression -> name`*. Right assignment is convenient when you type a complicated expression at the Spotfire S+ prompt and then realize you've forgotten to assign a name to the return value. Spotfire S+ also protects you from such forgetfulness by storing the last unassigned value in the `.Last.value` object in your working data directory. For consistency, we recommend that you always use left assignment within functions. If you use right assignment in a function definition and then view the code later, you will see that Spotfire S+ automatically reformats the function to use left assignment.

The *permanent assignment operator* `<<-` operator is like `<-`, except that it always writes to the working directory. Thus, it allows you to make permanent assignments from within functions. However, permanent assignment inside a function produces a side effect, in that objects in your working data directory are overwritten if they exist. This can lead to lost data. For this reason, we discourage the use of `<<-` within functions.

A more general form of assignment uses the `assign` function. The `assign` function allows you to choose where the assignment takes place. You can assign an object to either a position in the search list or a particular frame. For example, the following command assigns the value 3 to the name `boo` on the session frame 0:

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

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

Testing and Coercing Data

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

Unexpected data types can often cause a function to stop and return error messages. To protect against this behavior, many functions include expressions that test whether the input data is of the right type and coerce the data if necessary. For example, mathematical functions frequently have conditionals of the following form:

```
if(!is(x, "numeric")) x <- as(x, "numeric")
```

This statement tests the input data `x` with the `is` function. If `x` is not numeric, it is coerced to a numeric object with the `as` function.

As we discuss in Chapter 1, The S-PLUS Language, older versions of S-PLUS (S-PLUS 3.x, 4.x, and 2000) were based on version 3 of the S language (SV3). Most testing of SV3 objects is done with functions having names of the form `is.`*`type`*, where *`type`* is a recognized data type. For example, the functions `is.vector` and `is.matrix` test whether the data type of an object is a vector and a matrix, respectively. Functions also exist to test for special values such as `NULL` and `NA`; see the section Comparison and Logical Operators (page 164) for more information.

Coercion of SV3 objects can be performed using functions with names of the form `as.`*`type`*, such as `as.vector` and `as.matrix`. Coercion using the `as.`*`type`* functions is very strong, however, and can lead to loss of information; see the section Coercion of Values on page 66 for a full discussion. 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"
```

For a list of atomic modes, see the help file for the `mode` function.

Newer versions of S-PLUS (S-PLUS 5.x and later) are based on version 4 of the S language (SV4), which implements a vastly different approach to classes. In SV4, the `is.`*`type`* and `as.`*`type`* functions are collapsed into the simpler `is` and `as` functions. For example, to test whether an object `x` is numeric, type:

```
> is(x, "numeric")
```

Similarly, to coerce `x` to have a character data type, use the following command:

```
> as(x, "character")
```

The `is` and `as` functions are backwards compatible and can be used with data objects created in earlier versions of S-PLUS.

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

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

For information on classes and inheritance, see Chapter 12, Object-Oriented Programming in Spotfire S+.

Table 7.5 lists the most common testing and coercing functions. The first column gives the data type and the next two columns list the SV4 testing and coercing functions for the data type. The functions relating to the three data types `single`, `double`, and `integer` are used to modify the *storage mode* of numeric data. The storage mode of data is important if you need to interface with C or Fortran routines, but can safely be ignored otherwise.

Table 7.5: *Common functions for testing and coercing data objects.*

Type	Testing	Coercion
array	`is(x, "array")`	`as(x, "array")`
character	`is(x, "character")`	`as(x, "character")`
complex	`is(x, "complex")`	`as(x, "complex")`
data frame	`is(x, "data.frame")`	`as(x, "data.frame")`
double	`is(x, "double")`	`as(x, "double")`
factor	`is(x, "factor")`	`as(x, "factor")`
integer	`is(x, "integer")`	`as(x, "integer")`
list	`is(x, "list")`	`as(x, "list")`
logical	`is(x, "logical")`	`as(x, "logical")`
matrix	`is(x, "matrix")`	`as(x, "matrix")`
numeric	`is(x, "numeric")`	`as(x, "numeric")`
single	`is(x, "single")`	`as(x, "single")`
vector	`is(x, "vector")`	`as(x, "vector")`

OPERATING ON SUBSETS OF DATA

Often, we want to perform calculations on only a subset of a data set. The most useful method in S-PLUS for acting on a subset of data is called *subscripting*. In general, subscripting is good Spotfire S+ programming because it treats a data object as a whole rather than as a collection of elements. In fact, subscripting is much more powerful in S-PLUS than in other languages, and therefore should be mastered. For a collection of good Spotfire S+ programming techniques, see the section Organizing Computations (page 185).

In the following sections, we illustrate subscripting on a number of common data structures. For vectors, matrices, and arrays, we use square brackets `[]` to subset certain elements; for lists and data frames, we also use the dollar sign `$`.

Subscripting Vectors

A vector is a set of values that can be thought of as a one-dimensional array. (Note that this is simply a description, however; a S-PLUS vector is *not* equivalent to a S-PLUS one-dimensional array.) A vector subscript corresponds to an element's position, or *index*, in the vector. For example, the sixth element in a vector `x` has a subscript (or index) of 6. You can subscript a data vector by providing a set of indices that correspond to the elements you wish to keep. If `y` is a vector of indices, `x[y]` returns the elements in `x` that correspond to the indices.

In Spotfire S+, appropriate indices for subscripting vectors are constructed automatically from information supplied in one of the four following forms: a vector of positive integers, a vector of negative integers, a logical vector, and a vector of character strings. We discuss each of these in detail below. It is important to note that any S-PLUS expression that evaluates to an appropriate subscript value can be included in the square brackets. This flexibility makes subscripting a very powerful programming tool in Spotfire S+.

Subscripting with positive integers

If you supply a set of positive integers to subscript a vector, Spotfire S+ interprets the integers as the indices of the elements that you want to keep. To illustrate this, suppose we have a vector `x`:

```
> x <- c(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
[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
```

The following command returns the third element of `x`:

```
> x[3]
[1] 4.1
```

The next command returns the third, fifth, and ninth elements:

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

Note that the indices do not need to be unique:

```
> x[c(5,5,8)]
[1] 3.6 3.6 3.2
```

In addition, the indices do not need to be given in increasing order. Since `x` has twelve elements, the following returns `x` in reverse order:

```
> x[12:1]
[1] 1.0 7.4 7.6 6.6 3.2 2.8 2.3 3.6 2.6 4.1 3.0 1.9
```

To determine the total number of elements in a vector, use the function `length`. This function returns the number of elements in atomic objects such as vectors and matrices, and it returns the number of components in recursive objects such as lists. If the requested index for a vector `x` is greater than `length(x)`, Spotfire S+ returns `NA` to indicate a missing value.

Subscripting with negative integers

If you supply a set of negative integers to subscript a vector, Spotfire S+ interprets them as the indices of the elements that you want to exclude from the result. All elements in the original vector are returned in order, with the exception of those corresponding to the indices you specify. For example, the following command returns all elements in `x` except for the third, fourth, and fifth:

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

Specifying an index greater than `length(x)` has no effect:

```
> x[-13]
[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
```

The entire vector x is returned by this command, since x has only 12 elements.

Note that you cannot combine positive and negative integers to subscript a vector. For example, the command x[c(3,-5,9)] returns an error.

Subscripting with logical values

If you supply a set of logical values to subscript a vector, Spotfire S+ interprets the TRUE values as the indices of the elements that you want to keep. All elements in the original vector are returned in order, with the exception of those corresponding to the FALSE indices. For example, the commands below returns all elements in x that are greater than 2. Equivalently, they return all elements in x for which the vector x > 2 is TRUE:

```
> x > 2
[1] F T T T T T T T T T T F

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

The next command returns the elements in x that are between 2 and 4:

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

Logical index vectors are generally the same length as the vectors to be subscripted. However, this is not a strict requirement, as Spotfire S+ recycles the values in a short logical vector so that its length matches a longer vector. Thus, you can use the following command to extract every third element from x:

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

The index vector c(F,F,T) is repeated four times so that its length matches the length of x. Likewise, the following command extracts every fifth element from x:

```
> x[c(F,F,F,F,T)]
[1] 3.6 7.6
```

In this case, the index vector is repeated three times, and no values are returned for indices greater than `length(x)`.

Subscripting with character values

When you supply a set of character values to subscript a vector, the values must be from the vector's `names` attribute. Thus, this subscripting technique requires the vector to have a non-null `names` attribute. Spotfire S+ matches the names you specify with those in the `names` attribute, and returns the corresponding elements of the vector.

For example, consider the built-in vector `state.abb`, which contains the postal abbreviations for all fifty states in the USA. Note that `state.abb` has no names by default:

```
> length(state.abb)
[1] 50

> names(state.abb)
NULL
```

We can use the `"names<-"` replacement function to assign names to `state.abb`. The names we choose are located in the vector `state.name`, which contains the full names of all fifty states:

```
> length(state.name)
[1] 50

> state.name

 [1] "Alabama"        "Alaska"         "Arizona"
 [4] "Arkansas"       "California"     "Colorado"
 [7] "Connecticut"    "Delaware"       "Florida"
[10] . . .
```

Before modifying a built-in data object, we must create a local copy of it in our working directory:

```
> state.abb <- state.abb
> names(state.abb) <- state.name
```

Finally, we can subscript `state.abb` directly with character vectors. The following command returns the postal abbreviations of Alaska and Hawaii:

```
> state.abb[c("Alaska", "Hawaii")]
```

```
 Alaska Hawaii
   "AK"   "HI"
```

Subscripting Matrices and Arrays

Subscripting data sets that are matrices or arrays is very similar to subscripting vectors. In fact, you can subscript them exactly like vectors if you keep in mind that arrays are stored in *column-major order*. You can think of the data values in an array as being stored in one long vector that has a `dim` attribute to specify the array's shape. Column-major order states that the data values fill the array so that the first index changes the fastest and the last index changes the slowest. For matrices, this means that data values are filled in column-by-column.

For example, suppose we have the following matrix `M`:

```
> M <- matrix(c(12,1,19,15,9,14,6,2,11,10,7,19), nrow=3)
> M

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

We can extract the eighth element of `M` as follows:

```
> M[8]
[1] 2
```

This corresponds to the element in the second row and third column of `M`. When a matrix is subscripted in this way, the element returned is a single number without dimension attributes. Thus, Spotfire S+ does not recognize it as matrix.

Spotfire S+ also lets you use the structure of arrays to your advantage by allowing you to specify one subscript for each dimension. Since matrices have two dimensions, you can specify two subscripts inside the square brackets. The matrix subscripts correspond to the row and column indices, respectively:

```
> M[2,3]
[1] 2
```

As with vectors, array subscripts can be positive integers, negative integers, logical vectors, or character vectors if appropriate. The following command returns a 2×2 submatrix of `M`, consisting of the first and third rows and the second and fourth columns:

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

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

The next command returns values from the same two columns, including all rows except the first:

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

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

The next example illustrates how you can use a logical vector to subscript a matrix or array. We use the built-in data matrix `state.x77`, which contains demographic information on all fifty states in the USA. The third column of the matrix, `Illiteracy`, gives the percent of the population in a given state that was illiterate at the time of the 1970 census. We first copy this column into an object named `illit`:

```
> dim(state.x77)
[1] 50  8

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

Next, we subscript `state.x77` on the `illit` values that are greater than two:

```
> 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
```

In the above command, the subscript `illit > 2` results in a logical value of length 50. The returned values are in rows for which `illit > 2` is `TRUE`, and are from the third, fourth, and fifth columns of `state.x77`.

It is also possible to subscript matrices and arrays by supplying character values that specify indices. The supplied values must be from the array's `dimnames` attribute. Spotfire S+ matches the names you specify with those in the `dimnames` attribute and returns the corresponding elements of the array. For example, the command below returns the element of `state.x77` in the row named `Arizona` and the column named `Area`:

```
> dimnames(state.x77)

[[1]]:
 [1] "Alabama"        "Alaska"         "Arizona"
 [4] "Arkansas"       "California"     "Colorado"
 [7] "Connecticut"    "Delaware"       "Florida"
[10] . . .

[[2]]:
[1] "Population" "Income"     "Illiteracy" "Life.Exp"
[5] "Murder"     "HS.Grad"    "Frost"      "Area"

> state.x77["Arizona", "Area"]
[1] 113417
```

Note that if the subscript for a given dimension is omitted, all subscripts are assumed. Thus, we can construct the `illit` object with the simpler command:

```
> illit <- state.x77[, 3]
```

As with vectors, array subscripts can be any expression that evaluates to an appropriate set of index values.

Dropping Indices from Arrays

By default, Spotfire S+ drops array dimensions whenever subscripting results in a lower-dimensional object. Thus, if you subscript a single column or a single value from a matrix, the returned object is a vector instead of a matrix. You can see this with the matrix `M` that we defined above:

```
> M[1,3]
```

```
[1] 6
```

To override this behavior, use the `drop` argument to the subscripting functions. By default, `drop=TRUE` and dimensions are dropped whenever possible. The command below sets `drop=FALSE` and thus keeps the matrix dimensions:

```
> K <- M[1, 3, drop=FALSE]
> K

     [,1]
[1,]    6

> is(K, "matrix")
[1] T

> dim(K)
[1] 1 1
```

Subscripting Arrays with Matrices

In general, operating on arrays of data is more complicated than operating on simple vectors. One problem, as we discuss above, is that subscripting can sometimes collapse your data. Another problem is that subscripting an *n*-dimensional array with *n* subscripts yields only rectangular data sets. Often, you need to extract more irregular subsets of arrays. You can do this by supplying a subscript matrix that represents the positions of the individual elements you wish to keep.

For example, suppose we want to extract two elements of `M`: the element in row 1 and column 2, and the element in row 3 and column 3. We can do this directly with the following command:

```
> c(M[1,2], M[3,3])
[1] 15 11
```

More generally, we 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

> M[subscr.mat]
[1] 15 11
```

Subscript matrices such as `subscr.mat` have as many columns as there are dimensions in the array. Each element you want to extract from the array corresponds to a row in the subscript matrix, and the entries in each row are the indices of the element.

Subscripting Lists

Lists are vectors of class `"list"` that can hold arbitrary S-PLUS objects as individual elements. For example:

```
> x <- c("Tom", "Dick", "Harry")
> mode(x)
[1] "character"

> mylist <- list(x = x)
> mylist[1]
$x:
[1] "Tom" "Dick" "Harry"

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

When it acts on vectors, the subscript operator `[]` returns a subvector that has the same mode as the original vector. Thus, `mylist[1]` is a list, just like the original object `mylist`. Yet the element `x` that we use to build `mylist` is of mode `"character"`. To extract the original structure of a list element, use the operator `[[]]`:

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

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

The subscript operator `[[]]` returns a single element of a vector; the mode of the element may be different than the mode of the original vector. Although it works on ordinary numeric and character vectors, the operator `[[]]` is most useful on lists. For this reason, we refer to it as the *list subscript operator.*

If the subscript for a list is itself a vector or a list, Spotfire S+ uses it recursively. That is, the first element of the subscript extracts an element from the top-level list in the object, the next subscript element extracts from the first, and so on. For example, consider the object `biglist` below:

```
> biglist <- list(
+ lista = list(
+     list1 = list(x=1:10, y=10:20),
+     list2 = letters),
+ listb = list("a", "r", "e"))

> biglist

$lista:
$lista$list1:
$lista$list1$x:
 [1]  1  2  3  4  5  6  7  8  9 10

$lista$list1$y:
 [1] 10 11 12 13 14 15 16 17 18 19 20


$lista$list2:
 [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"


$listb:
$listb[[1]]:
[1] "a"

$listb[[2]]:
[1] "r"

$listb[[3]]:
[1] "e"
```

Suppose we want to extract the element `y` from `biglist`. The `lista` element is the first in `biglist`, the `list1` element is the first in `lista`, and `y` is the second element in `list1`. Thus, the following expression returns the contents of `y`:

```
> biglist[[1]][[1]][[2]]
[1] 10 11 12 13 14 15 16 17 18 19 20
```

We can accomplish the same thing more compactly with the following shorthand:

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

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

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

```
> mylist$x
[1] "Tom" "Dick" "Harry"
```

Note that the component operator returns the original structure of a list element, just like the list subscript operator:

```
> mode(mylist$x)
[1] "character"
```

You can extract components of embedded lists with nested use of the component operator:

```
> biglist$lista$list2

 [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 supply a vector of component names to the list subscript operator. The effect is the same as supplying a vector of component numbers, as in the `biglist[[c(1,1,2)]]` command above. For example, the following extracts the `list2` component of `lista` in `biglist`:

```
> biglist[[c("lista","list2")]]

 [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 use them interchangeably, and you can use both operators in a single command:

```
> biglist[["lista"]]$list1

$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
```

Subscripting Data Frames

Data frames share characteristics of both matrices and lists. Thus, subscripting data frames shares characteristics of subscripting both matrices and lists. In the examples below, we illustrate the possible ways that you can use to subscript data frames.

First, we form a data frame from numerous built-in data sets that contain information on the 50 states in the USA:

```
> state.data <- data.frame(state.abb, state.center,
+    state.region, state.division, state.x77,
+    row.names = state.name)
```

The data frame `state.data` is a matrix with 50 rows and 14 columns, but Spotfire S+ also recognizes it as a list. For most subscripting purposes, it is easiest to treat data frames as matrices and extract elements with the matrix subscript notation. For example, the following command returns a 3×3 data frame using matrix subscripts:

```
> state.data[5:7, 6:8]

            Population Income Illiteracy
 California      21198   5114        1.1
   Colorado       2541   4884        0.7
Connecticut       3100   5348        1.1
```

Like lists, data frames also have components that you can access with the component operator. Data frame components are the named columns and can be accessed just like list components. For example, the following command returns the `Population` column of `state.data`:

```
> state.data$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
```

You can also supply a vector of component names to subscript particular columns from a data frame. The following command returns a data frame containing both the `Population` and `Area` columns from `state.data`:

```
> state.data[, c("Population", "Area")]

                Population   Area
        Alabama       3615  50708
         Alaska        365 566432
        Arizona       2212 113417
       Arkansas       2110  51945
     California      21198 156361
       Colorado       2541 103766
    Connecticut       3100   4862
    . . .
```

ORGANIZING COMPUTATIONS

As with any programming task, the key to successful Spotfire S+ programming is to organize your computations before you start. Break the problem into 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.

Spotfire S+ programming in particular requires one additional bit of wisdom that is crucial: *treat every object as a whole*. Treating objects as whole entities is the basis for vectorized computation. You should avoid operating on individual observations, as such computations in Spotfire S+ carry a high premium in both memory use and processing time. Operating on whole objects is made simpler by a very flexible subscripting capability, as we discuss in the previous section. In most cases where `for` loops (or other loop constructs) seem the most natural way to access individual data elements, you will gain significantly in performance by using some form of subscripting.

In this section, we provide some high-level suggestions for good Spotfire S+ programming style. In addition, we discuss common control functions such as `if`, `ifelse`, and `return`.

Programming Style

Most programmers have been exposed to some general rules of programming style before they attempt to write functions in the S-PLUS language. “Avoid Goto’s,” “Use top-down design,” and “Keep it modular” are some catch-phrases that embody style guidelines. Most of the style guidelines you’ve come to swear by are also applicable to Spotfire S+, including:

- *Modularize your code*. If you need to design a large function, see if you can use smaller functions to do most of the work. This reduces the size of the larger function, which makes it easier to understand and debug. This approach also allows each of the smaller functions to be reused 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 it may be completely opaque to anyone else who tries to read it.

The S-PLUS language designates comments with the pound sign #. Be sure to read the section Notes Regarding Commented Code (page 198) before including comments in your S-PLUS functions.

- *Document your code*. If you distribute your functions to other users, include help files describing them that contain 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 know of a function that performs a certain task, use it instead of rewriting it. Spotfire S+ includes over four thousand built-in functions, most of which can be used to good effect in your own functions.
- *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 functions. If you plan to maintain such functions, however, it is clearer to have the precedence explicit with parentheses. For details, see the section Function Names and Operators (page 153).
- *Avoid unnecessary looping*. As we mention above, a key point in Spotfire S+ programming is to treat data objects as whole objects. When you begin to use a `for` loop, ask yourself if the loop can be eliminated in favor of a single expression that operates on the whole object. For atomic objects such as vectors and matrices, this is almost always possible. Lists, however, sometimes require `for` loops to access the individual list elements. The `lapply` function performs this looping for you; you should use it instead of explicitly constructing your own loops. Chapter 14, Using Less Time and Memory, in the *Application Developer's Guide*, discusses several techniques for avoiding loops in S-PLUS code, including the use of `lapply`.

Many other useful programming rules can be found in introductory programming texts.

Flow of Control

Normally, S-PLUS expressions are evaluated sequentially. Groups of expressions can be collected within curly braces `{}`. Such groups are treated as a single S-PLUS expression, but within the expression evaluation again proceeds sequentially. You can override the normal flow of control with the constructions presented in Table 7.6. In the subsections that follow, we discuss each of these in detail.

Table 7.6: *S-PLUS constructions that allow you to override the normal flow of control.*

Construction	**Description**
`if(`*`condition`*`) {`*`expression`*`}`	Evaluates *`condition`*. If true, evaluates *`expression`*.
`if(`*`condition`*`) {`*`expression1`*`}` `else {`*`expression2`*`}`	Evaluates *`condition`*. If true, evaluates *`expression1`*. If false, evaluates *`expression2`*.
`ifelse(`*`condition`*`,` *`expression1`*`,` *`expression2`*`)`	Vectorized version of the `if` statement. Evaluates *`condition`* and returns elements of *`expression1`* for true values and elements of *`expression2`* for false values.
`switch(`*`expression`*`, ...)`	Evaluates *`expression`*, which must return either a `character` or `numeric` value. The value of *`expression`* is compared to the remaining arguments: if it matches one exactly, the value of the evaluated argument is returned.
`break`	Terminates the current loop and passes control out of the loop.
`next`	Terminates the current iteration of the loop and immediately starts the next iteration.
`return(`*`expression`*`)`	Terminates the current function and immediately returns the value of *`expression`*.
`stop(`*`message`*`)`	Signals an error by terminating evaluation of the current function, printing the character string contained in *`message`*, and returning to the Spotfire S+ prompt.

Table 7.6: *S-PLUS constructions that allow you to override the normal flow of control. (Continued)*

Construction	Description
`while(condition) {expression}`	Evaluates `condition`. If true, evaluates `expression` then repeats the loop, evaluating `condition` again.
`repeat {expression}`	Simpler version of the `while` statement. No tests are performed and `expression` is evaluated indefinitely. Because `repeat` statements have no natural termination, they should contain `break`, `return` and/or `stop` statements.
`for(name in expression1) {expression2}`	Evaluates `expression2` once for each `name` in `expression1`. Although `for` loops are widely used in most programming languages, they are generally less efficient in Spotfire S+ than vectorized calculations.

The if and stop Statements

The `if` statement is the most common branching construction in S-PLUS. The syntax is simple:

```
if(condition) { expression }
```

Here, *`condition`* is any S-PLUS expression that evaluates to a logical value, and *`expression`* is the expression that is evaluated if *`condition`* is true. As with function bodies, *`expression`* needs to be braced only if it contains multiple statements. We suggest, however, that you include braces at all times for consistency and maintainability.

You can use `if` statements to screen input data for suitability. For example, the following issues an error if the input data `x` is not numeric:

```
if (!is(x,"numeric"))
  stop("Data must be of mode numeric")
```

The `stop` function stops evaluation of the calling function at the point where `stop` occurs. It takes a single argument that should evaluate to a character string. If such an argument 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 yields the following output if `x` is not numeric:

```
Error: Data must be of mode numeric
Dumped
```

We discuss the `stop` function more in the section Error Handling (page 205).

Another common use of `if` statements is in missing-argument handling within function definitions:

```
if(missing(y)) y <- sqrt(x)
```

This statement uses the `missing` function to check whether the argument `y` is missing; if it is, the value `sqrt(x)` is assigned to `y`.

When constructing conditions within `if` statements, you may want to test multiple conditions at once. You can use the two conditional operators, `&&` and `||`, for logical AND and OR statements, respectively. The syntax is:

```
if(condition1 && condition2) { expression }
if(condition1 || condition2) { expression }
```

These operators evaluate only as far as necessary to return a value. For example, the `&&` operator first evaluates *`condition1`*. If it is true, then *`condition2`* is evaluated and the result is returned as the value of the condition statement. If *`condition1`* is false, however, *`condition2`* is not evaluated and Spotfire S+ returns `FALSE` for the entire statement. Similarly, the `||` operator evaluates only until it encounters a true statement and then returns `TRUE`. It returns `FALSE` only if every condition is false.

Do not confuse the vectorized AND and OR operators (`&` and `|`) with the conditional operators discussed here. The vectorized operators return vectors of logical values, while conditionals return a single logical value. For more details, see the section Comparison and Logical Operators (page 164).

Note

Spotfire S+ recognizes `NA` as a logical value, giving three possibilities for logical data: `TRUE`, `FALSE`, and `NA`. If an `if` statement encounters `NA`, the calling function terminates and returns a message of the following form:

```
Missing value where logical needed
```

Multiple Cases: The if and switch Statements

One of the most common uses of the `if` statement is to provide branching for multiple cases. S-PLUS has no formal "case" statement, so you often implement cases using the following general form:

```
if(case1) { expression1 }
  else if(case2) { expression2 }
  else if(case3) { expression3 }
  . . .
  else lastexpression
```

The idea is to identify each case in your function and have it correspond to exactly one `if` or `else` statement. Such a construction makes it easy to follow the cases and serves as a check that all cases are covered. As an example, the simple function below generates a set of random numbers from one of three distributions:

```
my.ran <- function(n, distribution, shape)
{
  #
  # A function to generate n random numbers.
  # If distribution="gamma", shape must be given.
  #
  if(distribution == "gamma") return(rgamma(n,shape))
  else if(distribution == "exp") return(rexp(n))
  else if(distribution == "norm") return(rnorm(n))
  else stop("distribution must be \"gamma\", \"exp\", or
     \"norm\"")
}
```

We must use the escape character `\` in the `stop` message so that the double quotes are recognized.

The `switch` function handles multiple cases in a slightly different way than the `if` statement. The `switch` function accepts as its first argument a S-PLUS expression that should evaluate to a character string or numeric value. If the first argument is a character string, it is compared to the remaining arguments and the value of the matched argument is returned. If the first argument is an integer in the range `1:nargs - 1`, the integer `i` corresponds to the i+1st argument in the list. Integers tends to hide the nature of the individual cases, however; character values require that you label each individual case.

For example, we can rewrite `my.ran` using `switch` as follows:

```
my.ran2 <- function(n, distribution, shape)
```

```
{
  switch(distribution,
      gamma = rgamma(n,shape),
      exp = rexp(n),
      norm = rnorm(n),
      stop("distribution must be either \"gamma\",
         \"exp\", or \"norm\""))
}
```

When the `my.ran2` function is called, the interpreter evaluates the `distribution` argument. If it is one of the three character strings `"gamma"`, `"exp"`, or `"norm"`, the corresponding expression is evaluated. Otherwise, the `stop` expression is evaluated.

The ifelse Statement

The `ifelse` statement is a vectorized version of the `if` statement. The syntax is:

```
ifelse(condition, expression1, expression2)
```

The `ifelse` function evaluates both *`expression1`* and *`expression2`* and then returns the appropriate values from each based on the value of *`condition`*. If an element of *`condition`* is true, *`ifelse`* returns the corresponding value of *`expression1`*; otherwise, it returns the corresponding value of *`expression2`*.

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 needs to take place inside a loop. For example, here is one implementation of the built-in `sign` function, which accepts a numeric object and returns ± 1 depending on the sign of the elements:

```
my.sign <- function(x)
{
  for(i in 1:length(x)) {
      if(x[i] > 0) { x[i] <- 1 }
      else if(x[i] < 0) { x[i] <- -1 }
  }
  return(x)
}
```

The `ifelse` function provides a method for evaluating a condition over an entire vector or array of values without resorting to a `for` loop. Here is a rewritten version of the `my.sign` function that uses `ifelse` twice:

```
my.sign2 <- function(x)
{
  ifelse(x > 0, 1, ifelse(x < 0, -1, 0))
}
```

Not only is the version using `ifelse` much quicker, but it also handles missing values:

```
> my.sign(c(1, 3, NA, -2))

Problem in my.sign: Missing value where logical needed:
if(x[i] > 0) { x[i] <- 1}
else if(x[i] < 0) { x[i] <- -1}

> my.sign2(c(1, 3, NA, -2))

[1] 1 1 NA -1
```

The `ifelse` function essentially uses subscripting, but includes some extra steps so that it behaves correctly with `NA` values:

```
> ifelse

function(test, yes, no)
{
  answer <- test
  test <- as.logical(test)
  n <- length(answer)
  if(length(na <- which.na(test)))
     test[na] <- F
  answer[test] <- rep(yes, length.out = n)[test]
  if(length(na))
     test[na] <- T
  answer[!test] <- rep(no, length.out = 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, while replacing the elements for which the test is false with another value. The `ifelse` function sets the subscripts corresponding to missing values to FALSE before replacing the true elements, thus avoiding the error about missing values that my.sign reports. It then resets those subscripts to TRUE before replacing the false elements. The net result is that missing values remain missing.

Warning

Note from the code above that ifelse subscripts using single numeric indices. Thus, it designed to work primarily with vectors and, as an extension, matrices. If you subscript a data frame with a single index, Spotfire S+ treats the data frame as a list and returns an entire column; for this reason, you should exercise care when using ifelse with data frames. For details on subscripting, see the section Operating on Subsets of Data (page 172).

If our original data have no missing values, we can improve the my.sign2 function further, using the original for loop in my.sign as a hint. The telltale construction x[i] <- indicates that we can try subscripting directly:

```
my.sign3 <- function(x)
{
  x[x > 0] <- 1
  x[x < 0] <- -1
  return(x)
}
```

For more hints on replacing for loops, see Chapter 14, Using Less Time and Memory, in the *Application Developer's Guide.*

The break, next and return Statements

It is often either necessary or prudent to leave a loop before it reaches its natural end. This is imperative in the case of a repeat statement, which has no natural end. In Spotfire S+, you exit loops using one of three statements: break, next, and return. Of these, return exits not only from the current loop, but also from the current function. The break and next statements allow you to exit from loops in the following ways:

- The break statement tells Spotfire S+ to exit from the current loop and continue processing with the first expression following the loop.

- The `next` statement tells Spotfire S+ to exit from the current iteration of the loop and continue processing with the next iteration.

For example, the function below 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 return all the aces.

```
draw.aces <- function()
{
  ndraws <- 0
  aces.drawn <- rep(F,4)
  repeat {
      draw <- sample(1:52, 1, replace=T)
      ndraws <- ndraws + 1
      if(draw %% 13 != 1)
          next
      aces.drawn[draw %/% 13 + 1] <- T
      if(all(aces.drawn))
          break
  }
  cat("It took", ndraws,
      "draws to draw all four of the aces!\n")
}
```

The repeat Statement

The `repeat` statement is the simplest looping construction in Spotfire S+. It performs no tests, but simply repeats a given expression indefinitely. Because of this, the repeated expression should include a way out, typically using either a `break` or `return` statement. The syntax for `repeat` is:

```
repeat { expression }
```

For example, the function below uses Newton's method to find the positive, real *j*th roots of a number. A test for convergence is included inside the loop and a `break` statement is used to exit from the loop.

```
newton <- function(n, j=2, x=1)
{
  #
  # Use Newton's method to find the positive, real
  # jth root of n starting at old.x == x.
  # The default is to find the 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)))
     # Compute relative error as a 2-norm.
     conv <- sum((new.x - old.x)^2 / old.x^2)
     if(conv < 1e-10)
        break
     old.x <- new.x
  }
  return(old.x)
}
```

The following command finds the square roots of the integers 4 through 9:

```
> newton(4:9)
[1] 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000
```

To condense the code, we can replace the `break` statement inside the loop with a `return` statement. This makes it clear what the returned value is and avoids the need for any statements outside the loop:

```
newton2 <- function(n, j=2, x=1)
{
  old.x <- x
  repeat
  {
     new.x <- old.x-((old.x^j-n) / (j * old.x^(j-1)))
     conv <- sum((new.x - old.x)^2 / old.x^2)
     if(conv < 1e-10)
        return(old.x)
     old.x <- new.x
  }
}
```

Of course, such an abrupt departure from the function is undesirable if additional calculations remain after the loop.

Note

The `newton` function is vectorized, as most S-PLUS functions should be. Thus, the convergence criteria given above is not ideal for Newton's method, since it does not check the convergence of individual values. The code is provided here to illustrate the `repeat` and `break` statements; if you wish to use the code in your work, you may want to experiment with different convergence conditions.

The while Statement

You use the `while` statement to loop over an expression until a true condition becomes false. The syntax is simple:

```
while(condition) { expression }
```

For example, the function below returns a vector that corresponds to the binary representation of an integer.

```
bitstring <- function(n)
{
  tmp.string <- numeric(32)
  i <- 0
  while(n > 0) {
      tmp.string[32-i] <- n %% 2
      n <- n %/% 2
      i <- i + 1
  }
  firstone <- match(1, tmp.string)
  return(tmp.string[firstone:32])
}
```

In the `bitstring` code, `n` is made smaller in each iteration and eventually becomes zero. We have no way of knowing beforehand exactly how many times we need to execute the loop, so we use `while`. Here is the result of calling `bitstring` with `n=13`:

```
> bitstring(13)
[1] 1 1 0 1
```

Note that the `bitstring` function is not vectorized. It accepts a single integer value and does not work when the argument `n` is a numeric vector.

Like the `for` statement, `while` is familiar to most programmers with experience in other languages. And, like the `for` statement, it can often be avoided in Spotfire S+ programming. You may need to use `while` or `for` as a last resort in Spotfire S+, but you should always try a vectorized approach first.

The for Statement

Using `for` loops is a traditional programming technique that is fully supported in Spotfire S+. Thus, you can translate most Fortran-like DO loops directly into S-PLUS `for` loops and expect them to work. However, as we have stated, using `for` loops in S-PLUS is usually not a good technique because loops do not treat data objects as whole objects. Instead, they attack the individual elements of data objects, which is often a less efficient approach in Spotfire S+. You should always be suspicious of lines in S-PLUS functions that have the following form:

```
x[i] <- expression
```

Code with this structure can usually be implemented more efficiently with subscripting.

The syntax of S-PLUS `for` loops is:

```
for(name in expression1) { expression2 }
```

S-PLUS evaluates *`expression2`* once for each *`name`* in *`expression1`*, where *`expression1`* evaluates to a vector. For example:

```
for(i in 1:10) print(i)
```

The index variable (`i` in the above example) has scope only within the body of the `for` loop.

Note that there are certain situations in which `for` loops may be necessary in Spotfire S+:

- when the calculation on the $i+1$st element in a vector or array depends on the result of the same calculation on the ith element.
- for some operations on lists. The `lapply` and `sapply` functions perform some looping implicitly and may be more efficient than loops you code yourself.

Notes Regarding Commented Code

Comments within S-PLUS functions are sometimes roughly handled by the interpreter. This is because Spotfire S+ attaches comments to the beginning of the expressions that follow them. If no expression follows a comment, it is not attached and will not be printed when you view the function. For example, suppose we define a function `primes` as follows:

```
primes <- function(n = 100)
{
  n <- as(abs(n), "integer")
  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
  }
  return(c(smallp, p))
}
```

If we type `primes` at the Spotfire S+ prompt, all of the function code is printed, including the comment. However, suppose we add another comment after the last `return` statement when we define the function:

```
  return(c(smallp, p)) # return the prime values
```

No expression follows this comment, so it is not printed when we view the code of `primes` at the Spotfire S+ prompt.

SPECIFYING ARGUMENT LISTS

A well-chosen argument list can add considerable flexibility to most functions. Some languages, notably C, make a distinction between a function's *parameter list* and a function call's *argument list*. Spotfire S+ maintains this distinction by speaking of an argument's *formal name*, which corresponds to the name specified in a parameter list, and its *actual name*, which is used when actually calling the function. In this section, we present many examples of argument lists in S-PLUS functions and give suggestions for constructing your own.

Formal and Actual Names

When you define a S-PLUS function, you specify the arguments the function accepts by means of *formal names*. Formal names can be any combination of letters, numbers, and periods, as long as they are syntactically valid and do not begin with a number. The formal name `...` (three dots) is used to pass arbitrary arguments to a function; we discuss this in the section Variable Numbers of Arguments (page 202).

For example, consider the argument list of the `hist` function:

```
> args(hist)

function(x, nclass = "Sturges", breaks, plot = TRUE,
  probability = FALSE, include.lowest = T, ...,
  xlab = deparse(substitute(x)))
```

The formal names for this argument list are `x`, `nclass`, `breaks`, `plot`, `probability`, `include.lowest`, `...`, and `xlab`.

When you call a function, you specify *actual names* for each argument. Unlike formal names, an actual name can be any valid S-PLUS expression that makes sense to the function. You can thus provide a function call such as `length(x)` as an argument. For example, suppose we want to create a histogram of the `Mileage` column in the `fuel.frame` data set:

```
> hist(fuel.frame$Mileage)
```

The expression `fuel.frame$Mileage` is the actual name that corresponds to the formal argument `x`.

Specifying Default Arguments

In general, there are two ways to specify default values for arguments in a S-PLUS function:

- The simplest way is to use the structure `formalname=value` when defining a formal argument. For example, consider again the argument list for the `hist` function.

  ```
  > args(hist)
  function(x, nclass = "Sturges", breaks, plot = TRUE,
     probability = FALSE, include.lowest = T, ...,
     xlab = deparse(substitute(x)))
  ```

  Default values are supplied for the `nclass`, `plot`, `probability`, `include.lowest`, and `xlab` arguments.

- You can also specify defaults by providing code in the body of a function that handles missing arguments. This technique is useful if the code for computing a default value is too complicated to include in the formal argument list. We discuss this more in the next section.

Handling Missing Arguments

To test whether a given argument is supplied in the current function call, use the construction `if(missing(formalname))`. For example, the following code sample from `hist` shows how it handles a missing `breaks` argument:

```
if(missing(breaks)) {
  if(is.character(nclass))
     nclass <- switch(casefold(nclass),
        sturges = nclass.sturges(x),
        fd = nclass.fd(x),
        scott = nclass.scott(x),
        stop("Nclass method not recognized"))
  else if(is.function(nclass)) nclass <- nclass(x)
  breaks <- pretty(x, nclass)
  if(length(breaks) == 1) {
     if(abs(breaks) < .Machine$single.xmin * 100)
        breaks <- c(-1, -0.5, 0.5, 1)
     else if(breaks < 0)
        breaks <- breaks * c(1.3, 1.1, 0.9, 0.7)
     else
        breaks <- breaks * c(0.7, 0.9, 1.1, 1.3)
  }
```

```
    if((!include.lowest && any(
        x <= breaks[1])) || any(x < breaks[1]))
            breaks <- c(breaks[1] - diff(breaks)[1], breaks)
    x[x > max(breaks)] <- max(breaks)
  }
```

The construction `if(missing(`*`formalname`*`))` is useful for specifying a default value if the code for computing the default is too complicated to include in the formal argument list. Otherwise, the construction *`formalname`*`=`*`value`* is usually simpler.

Lazy Evaluation

Many programmers with experience in other programming languages make too much use of missing-argument handling in Spotfire S+. This is because Spotfire S+ uses *lazy evaluation*, which means that arguments are evaluated only as needed.

For example, consider the following simple plotting function:

```
  plotsqrt <- function(x,y)
  {
    z1 <- seq(1,x)
    if(missing(y)) plot(z1, sqrt(z1))
    else plot(z1,y)
  }
```

In this function, the missing-argument construction supplies the default value `sqrt(z1)` for the argument `y`. The default depends on the value `z1`, which is unknown until the completion of the first line in the body of the function. Because of this, many programmers avoid defining the default in the formal argument list. However, lazy evaluation allows us to do this in Spotfire S+ without receiving an error. Thus, we can rewrite `plotsqrt` as follows:

```
  plotsqrt2 <- function(x, y=sqrt(z1))
  {
    z1 <- seq(1,x)
    plot(z1,y)
  }
```

Spotfire S+ doesn't need the value for `y` until the final expression, at which time it can be successfully evaluated. In many programming languages, such a function definition causes errors similar to `Undefined variable sqrt(z1)`. In Spotfire S+, however, arguments aren't evaluated until the function body requires them.

Variable Numbers of Arguments

When you write functions for custom graphics or statistical procedures, you often build on existing functions that have large numbers of arguments. Frequently, you need only a few new arguments for your particular purpose. You can define only the arguments you need, but this reduces flexibility by limiting your access to the underlying function. You can specify defaults in the new function that cover every argument of the underlying function, but this is a burden during programming. Instead, you can use the special formal name `...` (three dots) to specify an arbitrary number of arguments.

In the section Specifying Default Arguments (page 200), we saw one example of the `...` argument in the `hist` function. The `hist` function is a special-purpose variant of the general function `barplot`, which accepts a large number of arguments. Rather than duplicate all of the `barplot` arguments, `hist` uses `...` to pass any the user specifies directly to `barplot`.

Within the body of a function, the only valid use of `...` is as an argument inside a function call. In the following code fragment from `hist`, the `...` argument passes all unmatched arguments from `hist` directly to `barplot`:

```
if(plot)
  invisible(barplot(counts, width = breaks,
     histo = T, ..., xlab = xlab))
```

The `counts`, `breaks`, and `xlab` objects are generated in the `hist` code and passed to the formal arguments in `barplot`. In addition, anything the user specifies that is not an element of the `hist` argument list is given to `barplot` through the `...` argument.

In general, arbitrary arguments can be passed to any function. You can, for example, create a function that computes the mean of an arbitrary number of data sets using the `mean` and `c` functions as follows:

```
my.mean <- function(...) { mean(c(...)) }
```

As a variation, you can use the `list` function to loop over arguments and compute the individual means of an arbitrary number of data sets:

```
all.means <- function(...)
{
  dsets <- list(...)
  n <- length(dsets)
  means <- numeric(n)
  for(i in 1:n) means[i] <- mean(dsets[[i]])
  return(means)
}
```

Note that formal arguments can follow `...` in function definitions. This construction is useful for functions such as `my.mean` and `all.means`, which compute a return value from an arbitrary number of data sets. To distinguish them from the data used to compute return values, arguments that follow `...` must be supplied by name when included in a function call and they cannot be abbreviated. For example, suppose we want to include the `trim` argument to `mean` in the `my.mean` function. We can do this with the following function definition:

```
my.mean <- function(..., trim=0.0)
{
  mean(c(...), trim=trim)
}
```

When calling `my.mean`, we can use the `trim` argument only by explicitly naming it:

```
> my.mean(corn.rain, corn.yield, trim=0.5)
[1] 17.95
```

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 name not matched
```

Required and Optional Arguments

Required arguments are those for which a function definition provides neither a default value nor missing-argument instructions. All other arguments are *optional*. For example, consider again the argument list for `hist`:

```
> args(hist)

function(x, nclass = "Sturges", breaks, plot = TRUE,
  probability = FALSE, include.lowest = T, ...,
    xlab = deparse(substitute(x)))
```

Here, `x` is a required argument. The `breaks` argument is optional because code is included in the body of `hist` to handle the case when `breaks` is missing. The `nclass`, `plot`, `probability`, `include.lowest`, and `xlab` arguments are optional with defaults defined in the argument list. The `...` argument allows you to pass other arguments directly to the `barplot` function. For information on defining defaults, see the section Specifying Default Arguments (page 200).

To see a function's required and optional arguments without viewing Spotfire S+ code, see the on-line help. The `hist` help file, for example, lists `x` as the only required argument; the remaining arguments are all listed as optional.

ERROR HANDLING

An often neglected aspect of function writing is *error-handling*, in which you specify what to do if something goes wrong. When writing quick functions for your own use, it doesn't make sense to invest much time in "bullet-proofing" your functions: that is, in testing the data for suitability at each stage of the calculation and providing informative error messages and graceful exits from the function if the data proves unsuitable. However, good error handling becomes crucial when you broaden the intended audience of your function.

In the section Flow of Control (page 186), we saw one mechanism in `stop` for implementing graceful exits from functions. The `stop` function immediately stops evaluation of the current function, issues an error message, and then dumps debugging information to a data object named `last.dump`. The `last.dump` object is a list that can either be printed directly or reformatted using the `traceback` function. For example, here is the error message and debugging information returned by the `my.ran` function from page 190:

```
# Call my.ran with an unrecognized distribution.
> my.ran(10, distribution="unif")

Problem in my.ran(10, distribution = "unif"): distribution
must be "gamma", "exp", or "norm"
Use traceback() to see the call stack

> traceback()

6: eval(action, sys.parent())
5: doErrorAction("Problem in my.ran(10, distribution =
\"unif\"): distribution must be \"gamma\", \"exp\", or
\"norm\"",
4: stop("distribution must be \"gamma\", \"exp\", or
\"norm\"")
3: my.ran(10, distribution = "unif")
2: eval(expression(my.ran(10, distribution = "unif")))
1:
Message: Problem in my.ran(10, distribution = "unif"):
distribution must be "gamma", "exp", or "norm"
```

The amount of information stored in `last.dump` is controlled by the `error` argument to the `options` function. The default value is `dump.calls`:

```
> options()$error
expression(dump.calls())
```

The `dump.calls` function stores a list of function calls, starting with the top-level call and including all calls within the function up to and including the one that produced the error. Another option, the `dump.frames` function, provides more information because it includes the complete set of frames created during the evaluation. However, `dump.frames` can generate a very large `last.dump` object; it should therefore be used only for debugging purposes and not for general error-handling. Other possibilities for the `error` argument to `options` are discussed in Chapter 11, Debugging Your Functions.

It is good programming practice to place `stop` statements within functions to mark the limits of the function's capability. For example, we can rewrite our `newton2` function so that it stops evaluation if there are no real roots to compute:

```
newton3 <- function(n, j=2, x=1)
{
  if(n < 0 && j %% 2 == 0)
     stop("No real roots")
  old.x <- x
  repeat
  {
     new.x <- old.x-((old.x^j-n) / (j * old.x^(j-1)))
     conv <- sum((new.x - old.x)^2 / old.x^2)
     if(conv < 1e-10)
        return(old.x)
     old.x <- new.x
  }
}
```

The `warning` function is similar to `stop`, but does not cause Spotfire S+ to stop evaluation. Instead, Spotfire S+ continues evaluating after the warning message is printed to the screen. This is a useful technique for warning users about potentially hazardous conditions such as data coercion:

```
if (!is(x, "numeric")) {
    warning("Coercing to mode numeric")
    x <- as(x, "numeric")
  }
```

As with most matters of programming style, the degree to which you incorporate stops and warnings depends on the level of finish you intend for your functions. Functions for distribution to others should be held to a higher standard than functions for your own use.

INPUT AND OUTPUT

Data Input

Most data input to S-PLUS functions is in the form of named objects passed as required arguments to the functions. For example:

```
> mean(corn.rain)
[1] 10.78421
```

Data can also be generated "on-the-fly" by passing S-PLUS expressions as arguments, such as calls to the `c` function:

```
> mean(c(5,9,23,42))
[1] 19.75
```

However, if you build turnkey systems or other applications in which you want to hide as much of the Spotfire S+ machinery as possible, your needs may go beyond this. Instead, you might want to build functions that read data from an existing file, create a S-PLUS object from the data, perform some analysis, and then return a value. Such functions conceal much of the structure of S-PLUS objects from users who may not know (or care to know) such details.

The principal tools for reading data from files are `scan`, `read.table`, and `importData`. The `scan` function reads ordinary sequential text files, the `read.table` function imports tabular text data into S-PLUS data frames, and `importData` reads data from a number of different file formats. Chapter 9, Importing and Exporting, discusses the three functions in detail.

Data Output

Spotfire S+ is an interactive system, so virtually anything you type prompts a response from Spotfire S+. In general, this response is the value of the evaluated expression, which Spotfire S+ prints automatically. If the value is assigned, however, automatic printing is not performed:

```
> 7 + 3
[1] 10

> a <- 7 + 3
```

Other responses from Spotfire S+ range from error messages to interactive prompts within a function call. We discuss error messages in the section Error Handling (page 205). The following subsections discuss four direct forms of creating output: return values, side effects, permanent data files, and temporary files.

Formatting Output

The format of printed return values in Spotfire S+ is determined partially by the mode of a returned object and partially by various session options. The examples below discuss different session options you can use to format output from your functions.

The width and length options

The `width` argument to the `options` function specifies the number of characters that fit on a line of output. By default, `width=80`:

```
> options()$width
[1] 80
```

The `length` argument to the `options` function specifies the number of lines that fit on a page of output. The `length` option also indicates where Spotfire S+ places `dimnames` attributes when a large matrix is printed; Spotfire S+ prints `dimnames` once on each page of output, and the `length` option governs how much information fits on a page. By default, `length=48`:

```
> options()$length
[1] 48
```

For example, if you use Spotfire S+ in a standard 80×24 terminal on a UNIX® system, the following call to `options` sets up your session so that long data sets can be viewed conveniently with the `page` function:

```
> options(length=23)
```

The digits option

The `digits` argument to the `options` function specifies the number of significant digits to print. By default, `digits=7`. To see full double precision output, set `digits=17` as follows:

```
> options(digits=17)
> pi
[1] 3.1415926535897931
```

Most print methods include a digits argument that can be used to override the value of options()$digits. Thus, you can call print explicitly with the desired number of significant digits. For example:

```
# Reset the digits option to its default.
> options(digits=7)

> print(pi, digits=17)
[1] 3.1415926535897931
```

On Windows®, you can also change the digits value through the **General Settings** dialog; select **Options** ► **General Settings** and click on the **Computations** tab to see this. It is important to note that any option changed through the GUI persists from session to session. In contrast, options changed via the options function are restored to their default values when you restart Spotfire S+. For more details, see the help files for the options function and the **Command Line Options** dialog.

The format, round, and signif functions

To print numeric data as a formatted character string, use the format function. This function returns a character vector the same length as the input in which all elements have the same length. The length of each element in the output is usually determined by the digits option. For example, the following command uses the default digits value of 7 to format and print the vector sqrt(1:10):

```
> 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"
```

Alternatively, we can set digits=3 as follows:

```
> 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"
```

The format function also includes a digits argument that can be used to override the value of options()$digits. The format function interprets digits as the number of significant digits retained, but it replaces trailing zeros with blanks:

```
# Reset the digits option to its default.
> options(digits=7)

> 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"
```

To include trailing zeros, you can use the `nsmall` argument to `format`, which sets the minimum number of digits to include after the decimal point:

```
> format(sqrt(1:10), digits=3, nsmall=2)

 [1] "1.00" "1.40" "1.73" "2.00" "2.24" "2.45" "2.64"
 [8] "2.83" "3.00" "3.16"
```

The `nsmall` argument is discussed in the help file for `format.default`.

You can use the `round` and `signif` functions to further control the action of the `digits` argument to `format`. The `round` function uses `digits` to specify the number of decimal places, while `signif` uses it to specify the number of significant digits retained. For example, note the difference in the output from the following two commands:

```
> 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"
```

Warning

If you want to print numeric values to a certain number of digits, do not use `print` followed by `round`. Instead, use `format` to convert the values to character vectors and then specify a certain number of entries. Printing numbers with `print` involves rounding, and rounding an already-rounded number can lead to anomalies. To see this, compare the output from the following two commands, for `x <- runif(10)`:

```
> round(print(x), digits = 5)
> as.numeric(format(x, digits = 5))
```

Note that the second command prints the correct number of digits but the first one doesn't.

This warning applies to all functions that use `print`, such as `var` and `cor`, and not just to the `print` function itself.

Constructing Return Values

When the body of a function is an expression enclosed in braces, the value of the function is the value of the last expression inside the braces. This fits well with the usual top-down design paradigm, where the goal is to start with some input, proceed through a set of operations, and return the finished output. For most simple functions, you need to verify only that the final value is what you actually want returned. Thus, if the body of a function carries out a series of replacements, the final line might be the name of the object in which the replacements were done. For example, the following function returns a modified version of the input object `x`:

```
bigger <- function(x,y)
{
  y.is.bigger <- y > x
  x[y.is.bigger] <- y[y.is.bigger]
  x
}
```

Even in simple functions like this, however, we recommend that you explicitly use a `return` statement to clearly identify the returned value:

```
bigger <- function(x,y)
{
  y.is.bigger <- y > x
  x[y.is.bigger] <- y[y.is.bigger]
```

```
    return(x)
  }
```

Often, you need to return a set of values that are generated throughout a function. To do this, assign the intermediate calculations to temporary objects within the function and then gather the objects 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 that month's sales using the daily sales information as the input data. Here is a function named `monthly.summary` that reads in such a data file, creates a matrix of the input data, and then performs the desired analysis:

```
monthly.summary <- function(datafile)
{
  x <- matrix(scan(datafile), nrow=10, byrow=T)
  store.totals <- rowSums(x)
  mean.sales <- mean(store.totals)
  attr(mean.sales, "dev") <- stdev(store.totals)
  best.performer <- match(max(store.totals), store.totals)
  return(list("Total Sales" = store.totals,
      "Average Sales" = mean.sales,
      "Best Store" = best.performer))
}
```

Notice that the function has no side effects. All calculations are assigned to objects in the function's frame, which are then combined into a list and returned as the value of the function. This is the preferred method for returning a number of different results in a S-PLUS function.

Suppose we have data files named `april.sales` and `may.sales` containing daily sales information for April and May, respectively. The following commands show how `monthly.summary` can be used to compare the data:

```
> 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

$"Best Store":
[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

$"Best Store":
[1] 6
```

As we discuss in the section Assignments (page 167), creating permanent objects from within functions is a dangerous practice because it can overwrite existing objects in your working directory. Thus, if our `monthly.summary` function creates permanent objects named `store.totals`, `mean.sales`, and `best.performer` instead of returning them as a list, we would lose the objects every time we ran the function. Instead, we recommend the list paradigm discussed above for returning a number of different results in a S-PLUS function.

Side Effects

A *side effect* of a function is any result that is not part of the returned value. Examples include graphics plots, printed values, permanent data objects, and modified session options or graphical parameters. Not all side effects are bad; graphics functions are written to produce side effects in the form of plots, while their return values are usually of no interest. In such cases, you can suppress automatic printing with the `invisible` function, which invisibly returns the value of a 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 .Call("s_pratom", x, TRUE, quote)
  invisible(x)
}
```

You should consciously try to avoid hidden side effects because they can wreak havoc with your data. Permanent assignment from within functions is the cause of most bad side effects. Many Spotfire S+ programmers are tempted to use permanent assignment because it allows expressions inside functions to work exactly as they do at the Spotfire S+ prompt. The difference is that if you type

```
myobj <<- expression
```

at the Spotfire S+ prompt, you are likely to be aware that `myobj` is about to be overwritten if it exists. In contrast, if you call a function that contains the same expression, you may have no idea that `myobj` is about to be destroyed.

Writing to Files

In general, writing data to files from within functions can be as dangerous a practice as permanent assignment. Instead, it is safer to create special functions that generate output files. Such functions should include arguments for specifying the output file name and the format of the included data. The actual writing can be done by a number of S-PLUS functions, the simplest of which are `write`, `write.table`, `cat`, `sink`, and `exportData`. The `write` and `write.table` functions are useful for retaining the structure of matrices and data frames, while `cat` and `sink` can be used to create free-format data files. The `exportData` function creates files in a wide variety of formats. See Chapter 9, Importing and Exporting, for details.

Functions such as `write`, `cat`, and `exportData` all generate files containing data; no S-PLUS structure is written to the files. If you wish to write the actual structure of your S-PLUS data objects to text files, use the `dump`, `data.dump`, or `dput` functions. We discuss each of these below.

The write and write.table functions

The `write` function writes S-PLUS vectors and matrices to specified files. It writes matrices column by column and includes five values in each line of the output file. For example, consider the following matrix, which we write to the output file **mat1.txt**:

```
> mat <- matrix(1:12, ncol=4)
> mat

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

> write(mat, file="mat1.txt")
```

Spotfire S+ stores **mat1.txt** in your working directory. You can view it in the text editor or pager of your choice. It contains the following three lines:

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

If you want to write a matrix to an output file in the same form as it appears in Spotfire S+, transform the matrix first with the `t` function and specify the number of columns with the `ncolumns` argument. For example:

```
> write(t(mat), file="mat2.txt", ncolumns=4)
```

The **mat2.txt** file looks similar to the object `mat`, and contains the following lines:

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

Alternatively, you can use the `write.table` function to write a vector, matrix, or data frame to a specified file. With `write.table`, you do not need to transpose the data object, and you can include row and column names in the output. For example, the following command creates a tab-delimited text file **fuel.txt** that contains the `fuel.frame` data set:

```
> write.table(fuel.frame, file = "fuel.txt", sep = "\t")
```

The cat and sink functions

The cat function is a general-purpose writing tool that can be used to write to the screen as well to files. The cat function is helpful for creating free-format data files, particularly when it is used with the format function. For example:

```
# Set the seed for reproducibility.
> set.seed(21)
> x <- runif(10)
> cat(format(x), fill=T)

0.8854639 0.3739834 0.4220316 0.2441107 0.6033186
0.5787458 0.3944685 0.5834372 0.1457345 0.4555785
```

The argument fill=T limits the width of each line in the output file to the width value specified in the options list. For more details on the format function and the width option, see the section Formatting Output (page 209).

To write to a file with cat, simply specify a file name with the file argument:

```
> cat(format(x), file="mydata1.txt")
```

Spotfire S+ stores **mydata1.txt** in your working directory. It overwrites any existing file named **mydata1.txt** unless you set the argument append=TRUE in the call to cat.

The sink function directs Spotfire S+ output into a file rather than to the screen. It can be used as an alternative to multiple cat(..., append=T) statements. For example, the following commands open a sink to a file named **mydata2.txt**, write x to the file in three different ways, and then close the sink so that Spotfire S+ writes future output to the screen:

```
> sink(file = "mydata2.txt")
> x
> format(x)
> format(x, digits=3)
> sink()
```

For more examples using sink, see the section Standard Connections (page 224).

The dump, data.dump, and dput functions

Files written by `cat` and `write` do not contain information regarding the structure of S-PLUS objects. To read the files back into S-PLUS objects, you must reconstruct this information. To write ASCII versions of S-PLUS objects that contain complete structural information, use the `dump`, `data.dump`, and `dput` functions.

The `dump` function is primarily a programmer's tool. It allows you to create editable, sourceable versions of S-PLUS functions. You can use `dump`, for example, to distribute a collection of functions via electronic mail. Alternatively, you can use `dump` to create a text file of a function, edit it outside of Spotfire S+, and then send the modified version to another user. To read your dumped functions back into Spotfire S+, use the `source` function.

The `data.dump` function writes S-PLUS data objects to files. It uses a text-based format so that the objects can be restored on any machine. Because of the special text format, you should not edit the files generated by `data.dump`. Instead, the primary uses of `data.dump` include transferring objects between versions of Spotfire S+, between machines, or between users. To read your dumped objects back into Spotfire S+, use the `data.restore` function.

Note

In earlier versions of S-PLUS, the `dump` function could be used to transfer data objects such as matrices and lists between machines. This behavior is no longer supported in SV4 versions of Spotfire S+. Currently, `dump` is used only for creating editable text files of S-PLUS functions; use `data.dump` to transfer your data objects between machines. For more details, see the help files for these two functions.

The `dput` function can be thought of as a companion to `assign`. Where `assign` creates S-PLUS objects in binary form, `dput` creates them in ASCII text. The output from `dput` can be read back into Spotfire S+ with the `dget` function.

Like `data.dump`, you can use the `dput` function to transfer objects between machines. However, the formats used by the two functions are slightly different. To see this, note the differences in the two files generated by the following commands:

```
# Set the seed for reproducibility.
```

```
> set.seed(49)
> tmp.df <- data.frame(x=1:10, y=runif(10))
> dput(tmp.df, file="mydata1.txt")
> data.dump("tmp.df", file="mydata2.txt")
```

The files **mydata1.txt** and **mydata2.txt** are stored in your working directory.

Files created by `data.dump` include the names of the dumped objects. Thus, you can read **mydata2.txt** into Spotfire S+ with `data.restore` and the object `tmp.df` becomes available in your working directory. In contrast, files created by `dput` include the contents of objects, but not the object names. The following commands illustrate this:

```
# Remove tmp.df and restore the contents
# of the file created by data.dump.
> rm(tmp.df)
> data.restore("mydata2.txt")
[1] "mydata2.txt"

> tmp.df

    x          y
 1  1 0.54033146
 2  2 0.27868110
 3  3 0.31963785
 4  4 0.26984466
 5  5 0.75784146
 6  6 0.32501004
 7  7 0.90018579
 8  8 0.04155586
 9  9 0.28102661
10 10 0.09519871

# Remove tmp.df and restore the contents
# of the file created by dput.
> rm(tmp.df)
> dget("mydata1.txt")

    x          y
 1  1 0.54033146
 2  2 0.27868110
 3  3 0.31963785
 4  4 0.26984466
 5  5 0.75784146
 6  6 0.32501004
 7  7 0.90018579
 8  8 0.04155586
 9  9 0.28102661
10 10 0.09519871

> tmp.df
Problem: Object "tmp.df" not found
```

You must assign the output from `dget` to access its contents in your working directory:

```
> tmp.df <- dget("mydata1.txt")
```

Creating Temporary Files

You can use `cat`, `write`, and `dput` together with the `tempfile` function to create temporary files that have unique names. Such files are convenient to use for a variety of purposes, including text processing tools. For example, the built-in `ed` function creates a temporary file that holds the object being edited on a Windows system:

```
> ed

function(data, file = tempfile("ed."),
  editor = "notepad", error.expr)
{
  drop <- missing(file)
  if(missing(data)) {
     if(!exists(".Last.file"))
        stop("Nothing available for re-editing")
     file <- .Last.file
     data <- .Last.ed
  }
  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)
. . .
```

If you are on a UNIX system, the syntax is slightly different:

```
> ed

function(data, file = tempfile("ed."),
  editor = "ed", error.expr)
{
  if(editor == "vi" && GUI.interactive()) {
     editor.ux = paste("xterm -e", editor)
  }
  else editor.ux = editor
  drop <- missing(file)
  if(missing(data)) {
     if(!exists(".Last.file"))
        stop("Nothing available for re-editing")
     file <- .Last.file
     data <- .Last.ed
  }
  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 for a temporary file. In the `ed` function above, the unique name is composed of the character string `ed.` and a unique ID number. Note that `tempfile` generates only a name for a temporary file and not the file itself. You must use `cat`, `write`, or `dput` to actually create and write to the file.

The temporary files created with `tempfile` are ordinary files written to the directory specified by the **S_TMP** environment variable. Customarily, this directory is a temporary storage location that is wiped clean frequently. To prevent overloading this directory, it is best if you incorporate file cleanup into your functions that utilize `tempfile`. This is discussed in the section Wrap-Up Actions (page 237). For more information on Spotfire S+ environment variables such as **S_TMP**, see Chapter 2, The Spotfire S+ Command Line and System Interface, in the *Application Developer's Guide.*

Connections

Connections are mechanisms for connecting Spotfire S+ to other processes in the computing environment. With connections, Spotfire S+ can efficiently and easily read or write streams of data. The most common example of a connection is a physical file; other examples include external processes that read or write data, and S-PLUS character vectors.

In general, connections provide several facilities for the Spotfire S+ programmer:

1. They provide a uniform mechanism for functions that need to read or write data.
2. They allow mixed reading and writing during a single Spotfire S+ session. Because Spotfire S+ manages all active connections, operations that are difficult or error-prone at a lower level are tractable with connections.
3. They hide many of the low-level programming details needed for doing input and output.

If you can express input and output computations in terms of connections, the result is usually convenient, reliable, and efficient code. In this section, we give a brief overview of Spotfire S+ connections; for further details and additional topics not discussed here, see Chapter 10 in Chambers (1998).

Connection Classes

Table 7.7 lists the connection classes available in Spotfire S+. Each of these classes extend the virtual class `"connection"`.

Table 7.7: *Classes of Spotfire S+ connections.*

Connection Class	Description
`file`	File connection. This is represented by a character vector naming the path of the file. If no path is supplied when the connection is opened, a temporary one is created.
`pipe`	System command, with standard input that Spotfire S+ can write to and standard output that Spotfire S+ can read from. This is represented by a character vector naming the command. Spotfire S+ opens the connection by executing the command; data written to the pipe then becomes its standard input.
`fifo`	First-in first-out connection. This is represented like a `file`, by a character vector naming a path. Unlike a `file`, a `fifo` holds on to data only until it is read, at which point the data effectively disappears. For this reason, a `fifo` is sometime referred to as a *named pipe*.
`textConnection`	Text connection. This is represented by a S-PLUS character vector. This class is provided mainly as a convenience, so that you can use objects containing character vectors in computations that expect to read from connections. By design, text connections are read-only in Spotfire S+.

All four classes listed in the table are functions that can be used to (optionally) open the described connections and return S-PLUS *connection objects*. Connection objects are one of the primary tools for managing connections in Spotfire S+. For example, the following command opens a file connection to **myfile.dat** and assigns the value to the connection object `filecon`.

```
> filecon <- file("myfile.dat")
```

The side effect of the call to `file` opens the connection, so you may be tempted to think that the returned object is of little interest. However, conscientious use of connection objects results in cleaner and more flexible code. For example, you can use these objects to delay opening particular connections. Each connection class has an optional argument `open` that can be used to suppress opening a connection. With the returned connection object, you can use the `open` function to explicitly open the connection when you need it:

```
# Create a file connection object but do not open it.
> textfile <- file("myfile.dat", open=F)

# After some time, open the connection.
> filecon <- open(textfile)

# After reading from or writing to the connection, close it.
> close(filecon)
```

Most S-PLUS connection functions abide by the following simple rules:

- If a connection is not currently open, open it and ensure the connection is closed at the end of the function call.
- If a connection is already open, leave it open at the end of the function call.

Thus, if you use one of the functions listed in Table 7.7 to open a connection, you do not need to explicitly close it. However, if you use the `open` function, you should ensure that the connection is properly closed by using the `close` function. Organizing your computations in this way prevents forgotten connections from consuming machine resources. See the section Support Functions for Connections (page 228) for additional tips on managing connections.

Standard Connections

Table 7.8 lists a number of predefined connections available in Spotfire S+. The `stdin`, `stdout`, and `stderr` functions organize user interactions into three traditional streams of data: input from the user, printed output, and errors or messages for the user. The function calls `stdin()`, `stdout()`, and `stderr()` return the current connections associated with standard input, standard output, and standard error, respectively. When Spotfire S+ is running as an interactive session, standard input is your keyboard and both standard output and standard error are your display.

Table 7.8: *S-PLUS functions associated with standard connections.*

Standard Connection	Description
`stdin`	Standard input.
`stdout`	Standard output.
`stderr`	Standard error. Spotfire S+ writes messages to this connection.
`auditConnection`	Connection to which Spotfire S+ writes auditing information about the session. See Chapter 13, The Spotfire S+ Command Line and the System Interface, in the *Application Developer's Guide* for information on the session audit file.
`clipboardConnection`	System clipboard for the Spotfire S+ session.
`sink`	Function that redirects the output associated with standard connections

You can redirect the output connection to another connection (usually a file) with the `sink` function. The `sink` command exists primarily for its side effect: sinked output remains redirected until another call to `sink` explicitly alters it. For example, the following commands redirect the standard output connection to the file **x.out**.

```
# Open the sink.
> sink("x.out")

# Generate 50 random numbers and print out
# their mean and variance.
> x <- runif(50)
> mean(x)
> var(x)
```

```
# Close the sink.
> sink()
```

Connection Modes

By default, `file`, `fifo`, and `pipe` connections are opened for both reading and writing, appending data to the end of the connection if it already exists. While this behavior is suitable for most applications, you may require different modes for certain connections. Example situations include:

- Opening a `file` connection as read-only so that it is not accidentally overwritten.
- Opening a `file` connection so that any existing data on it is overwritten, rather than appended to the end of it.

You can change the default mode of most connections through the `mode` argument of the `open` function. For example, the following commands open a file connection as write-only. If we try to read from the connection, Spotfire S+ returns an error:

```
# Create a file connection object but do not open it.
> textfile <- file("myfile.dat", open=F)

# Open the connection as write-only.
> filecon <- open(textfile, mode = "w")

> scan(filecon)
Problem in scanDefault(file, what, n): "myfile.dat" already
opened for "write only": use reopen() to change it

# Close the connection.
> close(filecon)
[1] T
```

As the error message suggests, you can use the `reopen` function to close the connection and reopen it with a different value for `mode`.

Note

The mode of a `textConnection` cannot be changed. By design, text connections are read-only.

Instead of explicitly calling `open`, you can supply the desired mode string to the `open` argument of one of the connection classes. Thus, the following command illustrates a different way of opening a file as write-only:

```
> filecon <- file("myfile.dat", open = "w")
```

Table 7.9 lists the most common mode strings used to open connections in Spotfire S+.

Table 7.9: *Common modes for Spotfire S+ connections.*

Mode String	Description
`"rw"`	Open for reading and writing, overwriting current data on the connection if it already exists.
`"ra"`	Open for reading and writing, appending data to the current version of the connection if it already exists. Writing is allowed only at the end of the connection.
`"r"`	Open for reading only.
`"w"`	Open for writing only, overwriting current data on the connection if it already exists.
`"a"`	Open for writing only, appending data to the current version of the connection if it already exists. Writing is allowed only at the end of the connection.
`"*"`	Open for writing only, appending data to the current version of the connection if it already exists. Writing is allowed anywhere in the connection; the initial write position is at the end.
`""`	Do not open the connection. Unopened connection objects can be opened explicitly at a later time using the `open` command.

Support Functions for Connections

The functions listed in the two tables below provide support for managing connections in your Spotfire S+ session: Table 7.10 describes functions that allow you to see any active connections and Table 7.11 describes functions that prepare connections for reading or writing. We have already seen the `open` and `close` functions in previous sections. In the text below, we describe each of the remaining support functions.

Table 7.10: *S-PLUS functions for managing active connections.*

Management Function	Description
`getConnection`	Returns the connection corresponding to the input argument.
`getAllConnections`	Returns a list of all open connections.
`showConnections`	Prints a table of all open connections.

Table 7.11: *Support functions that prepare connections for reading or writing.*

Support Function	Description
`open`	Open a connection explicitly for reading or writing.
`isOpen`	Check whether a connection is open.
`close`	Close a connection after reading from or writing to it.
`seek`	Position a file connection for reading or writing.

The getConnection and showConnections functions

You can view all active connections in your Spotfire S+ session by using the functions `getAllConnections` and `showConnections`. The `getAllConnections` function returns a list of all open connections that includes information on the class and mode of each. The showConnections function displays this information in a convenient tabular format. For example, suppose we open two connections to text files:

```
> filecon <- open("mydata.txt")
> filecon2 <- open("mydata2.txt", mode="w")
```

The showConnections function returns the following:

```
> showConnections()

    Class Mode   State   Description
52 "file" "*"   "Read"  "mydata.txt"
56 "file" "w"   "Write" "mydata2.txt"
```

The number at the beginning of each row in the table is a unique descriptor for the corresponding connection. We can use these numbers with the getConnection function to access individual connection objects. For example, the following command closes the connection to **mydata.txt**:

```
> close(getConnection(52))
[1] T
```

For file connections, we can also supply getConnection with a character string naming the file:

```
> close(getConnection("mydata2.txt"))
[1] T
```

The seek function

Spotfire S+ maintains separate positions on connections for reading and writing. Thus, you can write to a connection starting from a different location than the one used to read from the connection. Because the positions are separate, you may need explicit control over positioning in a connection object. The seek function allows you to do this with file connections. It accepts the following arguments:

- A file connection.
- The argument where, which is a position measured in bytes from the start of the file.
- The argument rw, which determines whether the "read" or "write" position is modified.

If where is given, seek moves the rw position to the specified value; otherwise, it returns the current rw position.

The following example from Chambers (1998) illustrates this function. Suppose an open file connection named f exists. We read one expression from it and then leave the connection so that reading begins again with the same expression.

```
# Return the reading position in the file.
> pos <- seek(f, rw = "read")

# Parse one expression.
> myexpr <- parse(f, n = 1)

# Reset the reading position.
> seek(f, where = pos, rw = "read")
```

For pipe and fifo connections, data is read in the same order in which it is written. Thus, there is no concept of a "read" position for these connections. Likewise, data is always written to the end of pipes and fifos, so there is also no concept of a "write" position. For textConnection objects, only "read" positions are defined.

Reading from and Writing to Connections

Table 7.12 lists the main S-PLUS functions for reading from and writing to connections. Wherever possible, we pair functions in the table so that relationships between the reading and writing functions are clear. For details on the scan, cat, data.restore, data.dump, source, dump, dget, and dput functions, see the section Writing to

Files (page 215). For details on `readRaw` and `writeRaw`, see the section Raw Data Objects (page 233). For examples using any of these functions, see the on-line help files.

Table 7.12: *S-PLUS functions for reading from and writing to connections. The first column in the table lists functions for reading; the second column lists the corresponding writing functions (if any).*

Reading Function	Writing Function	Description
`parse`		Read *n* S-PLUS expressions.
`parseSome`		Read *n* lines or 1 S-PLUS expression.
`scan`	`cat`	Read *n* data items. Write any number of data items.
`readLines`	`writeLines`	Read *n* lines and return one character vector per line. Write *n* lines, consisting of one character vector per line.
`read.table`	`write.table`	Read a two-dimensional table of data. Write a two-dimensional table of data.
`readRaw`	`writeRaw`	Read raw data objects. Write raw data objects.
`data.restore`	`data.dump`	Read dumped data objects. Write S-PLUS data objects to their dumped forms.
`source`	`dump`	Parse and evaluate *n* S-PLUS expressions. Write text representations of S-PLUS objects.
`dget`	`dput`	Read expressions that represent S-PLUS objects. Write expressions that represent S-PLUS objects.
`dataGet`	`dataPut`	Read the S-PLUS symbolic dump format. Write objects in the S-PLUS symbolic dump format.

Examples of Pipe Connections

The examples throughout most of this section deal mainly with `file` connections. This is because files are often the easiest of the connection classes to visualize applications for, while pipes and fifos tend to have more specialized applications. Here, we present three examples that illustrate how you might use pipe connections in your work.

Reading files compressed by gzip

The **gzip** program is a popular compression program that is distributed under the GNU public license. Binary versions of **gzip** are available from the Web site **http://www.gzip.org** for most flavors of UNIX®, Linux®, and Windows®.

Suppose you have a space-delimited collection of numbers stored in the file **primes.txt**:

```
2 3 5 7 11
13 17 19 23 29
31 37 41 43 47
53 59 61 67 71
73 79 83 89 97
```

To compress the file and write the results in **primes.gz**, issue the following system command:

gzip -c primes.txt > primes.gz

The following commands read the compressed file in Spotfire S+:

```
> p1 <- pipe("gzip -d -c primes.gz")
> scan(p1, sep=" ")

 [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61
[19] 67 71 73 79 83 89 97
```

Using Perl to remove trailing commas

Perl is powerful scripting language that is very good at manipulating text files. Binaries of Perl are freely available for nearly every operating system; you can find out more at **http://www.perl.com**.

The ability to write quick, one-line scripts in Perl makes it ideal for preprocessing data files through Spotfire S+ pipes. For example, suppose you have a comma-delimited data file named **comma.txt**:

```
8.4,2.8,2.0,4.2,
4.5,0.3,
8.1,7.3,0.4,
6.1,7.2,8.3,0.6,0.7,
3.7,
```

The process that generated the file placed a comma at the end of each line. If you use the `scan` function to read this file, S-PLUS includes an extra `NA` after each trailing comma. Instead, you can remove the trailing commas and read the data into Spotfire S+ as follows:

```
> p2 <- pipe('perl -p -e "s/,$//" comma.txt')
> scan(p2, sep=",")

 [1] 8.4 2.8 2.0 4.2 4.5 0.3 8.1 7.3 0.4 6.1 7.2 8.3 0.6
[14] 0.7 3.7
```

Using Perl to filter white space

Suppose you have a file named **white.txt** that contains white-space delimited numbers. Some of the white space may be tabs, some may be single spaces, and some might be multiple spaces:

```
4.02   4        2.03 1.62   4.67
2.15  2 4.83 4.87       2
4       4.38 1.83  4.38          4.73
4       4.28 5.45       1.77 4.22
```

Using Perl, you can replace the tabs and spaces between each pair of numbers with a single space. You can then read the file into Spotfire S+ by specifying a single white space as the delimiter. The following commands show how to do this:

```
> p3 <- pipe('perl -p -e "s/[\\ \\t]+/ /g" white.txt')
> scan(p3, sep=" ")

 [1] 4.02 4.00 2.03 1.62 4.67 2.15 2.00 4.83 4.87 2.00
[11] 4.00 4.38 1.83 4.38 4.73 4.00 4.28 5.45 1.77 4.22
```

Raw Data Objects

Raw data objects are structures that consist of undigested bytes of data. They can be thought of naturally as vectors of byte data. You can manipulate these objects in Spotfire S+ with the usual vector functions to extract subsets, replace subsets, compute lengths and define lengths. In addition, raw data can be passed as arguments to functions, included as slots or components in other objects, and assigned to any database. However, raw data objects are not are not numeric and cannot be interpreted as ordinary, built-in vectors.

Spotfire S+ provides no interpretation for the contents of the individual bytes: they don't have an intrinsic order, `NA`s are not defined, and coercion to numbers or integers is not defined. The only comparison operators that make sense in this setting are equality and inequality, interpreted as comparing two objects overall.

In Spotfire S+, raw data is usually generated in four basic ways:

1. Read the data from a file or other connection using the functions `readMapped` or `readRaw`. Conversely, you can write raw data to a file or connection using `writeRaw`.
2. Use character strings that code bytes in either hex or ascii coding. The character strings can then be given to the functions `rawFromHex` and `rawFromAscii` to generate the raw data.
3. Allocate space for a raw object and then fill it through a call to C code via the `.C` interface.
4. Call a Spotfire S+-dependent C routine through the `.Call` interface.

See Chapter 5, Interfacing with C and FORTRAN Code, in the *Application Developer's Guide* for details on `.C` and `.Call` interfaces. For details on additional topics not discussed here, see Chambers (1998).

The primary S-PLUS constructors for raw data are the `rawData` and `raw` functions. The four approaches mentioned above usually arise more often in practice, however. All raw data objects in S-PLUS have class `"raw"`, regardless of how they are generated.

Examples

```
# Generate raw data from an ascii character vector.
> rawFromAscii(letters[1:6])
rawData(6,c("64636261","6665"))

# Generate raw data from a hex-coded vector.
> rawFromHex(rep("3af", 4))
rawData(6,c("3aaff33a","aff3"))
```

Raw Data on Files and Connections

The `readMapped` function reads binary data of numeric or integer modes from a file. Typical applications include reading data written by another system or by a C or Fortran program. The function also provides a way to share data with other systems, assuming you know where the systems write data.

As its name suggests, `readMapped` uses memory mapping to associate the input file with a S-PLUS object, so the data is not physically copied. Therefore, the function is suitable for reading in large objects. The connection may be open in advance or not; in either case, `readMapped` never closes it since that invalidates the mapping. Note that you can open a file, position it with the `seek` function, and map the data starting from a position other than the beginning of the file. See the section Support Functions for Connections (page 228) for details on `seek`.

The `readRaw` function is like `readMapped`, but physically reads the data. Thus, it is suitable for connections that are not ordinary files and cannot be memory mapped. The `writeRaw` function is the counterpart to `readRaw`; it writes the contents of a S-PLUS object in raw form to either a file or a text connection. Only the data values are written, however. The resulting file does not include structural information, and any software that reads the values needs to know the type of data on the file.

Examples

The following example writes twenty integers to a raw data file named **x.raw**, and then reads the values back in using the `readRaw` function.

```
> x <- c(rep(5,5), rep(10,5), rep(15,5), rep(20,5))
> x
 [1]  5  5  5  5  5 10 10 10 10 10 15 15 15 15 15 20 20
[18] 20 20 20

> writeRaw(x, "x.raw")
NULL
```

To ensure the data are read into Spotfire S+ as integers, set the argument `what` to `integer()` in the call to `readRaw`:

```
> x1 <- readRaw("x.raw", integer())
> x1
```

```
 [1]  5  5  5  5  5 10 10 10 10 10 15 15 15 15 15 20 20
[18] 20 20 20
```

The next command reads only the first 10 integers into Spotfire S+:

```
> x2 <- readRaw("x.raw", integer(10))
> x2
 [1]  5  5  5  5  5 10 10 10 10 10
```

You can determine the amount of data that is read into Spotfire S+ in one of two ways: the `length` argument to `readRaw` or the length of the `what` argument. If `length` is given and positive, Spotfire S+ uses it to define the size of the resulting S-PLUS object. Otherwise, the length of `what` (if positive) defines the size. If `length` is not given and `what` has a length of zero, all of the data on the file or connection is read.

The following example writes twenty double-precision numbers to a raw data file named **y.raw**, and then reads the values back in using `readRaw`. Note that the values in the vector `y` must be explicitly coerced to doubles using the `as.double` function, so that Spotfire S+ does not interpret them as integers.

```
> y <- rep(as.double(1:5), times=4)
> writeRaw(y, "y.raw")
NULL
```

To ensure the data are read into Spotfire S+ as double precision numbers, set the argument `what=double()` in the call to `readRaw`:

```
> y1 <- readRaw("y.raw", double())
> y1
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

In contrast to many S-PLUS functions, it is the mode of the `what` argument that matters, not the class. Classes other than `"numeric"` (`integer` and `double`) might have a numeric prototype but `readRaw` works the same for all of them, reading in numeric values.

WRAP-UP ACTIONS

The more complicated your function, the more likely it is to complete with some loose ends dangling. For example, the function may create temporary files, or alter Spotfire S+ session options and 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 loose ends in your functions.

The most important wrap-up action is to ensure that a function returns the appropriate value or generates the desired side effect. Thus, the final line of a function is often the name of the object to be returned or an expression that constructs the object. See the section Constructing Return Values (page 212) for examples.

To restore session options or specify arbitrary wrap-up actions, use the `on.exit` function. With `on.exit`, you ensure the desired actions are carried out whether or not the function completes successfully. For example, highly recursive functions often overrun the default limit for nested expressions. The `expressions` argument to the `options` function governs this and is set to 256 by default. Here is a version of the factorial function that raises the limit from 256 to 1024 and then cleans up:

```
fac1024 <- function(n)
{
  old <- options(expressions = 1024)
  on.exit(options(old))
  if(n <= 1) { return(1) }
  else { n * Recall(n-1) }
}
```

The first line of `fac1024` assigns the old session options to the object `old`, and then sets `expressions=1024`. The call to `on.exit` resets the old options when the function finishes. The `Recall` function is used to make recursive calls in Spotfire S+.

Compare `fac1024` with a function that uses the default limit on nested expressions:

```
fac256 <- function(n)
{
  if(n <= 1) { return(1) }
```

```
    else { n * Recall(n-1) }
  }
```

Here is the response from Spotfire S+ when each function is called with `n=80.0`:

```
> fac1024(80.0)
[1] 7.156946e+118

> fac256(80.0)

Error: Expressions nested beyond limit (256)
  only 30 of 110 frames dumped
only the first of 10 elements used for string value
```

Note

As defined, the `fac1024` function must be called with a real argument such as 80.0. If you call it with an integer such as 80, Spotfire S+ overflows and returns `NA`. See the section Integer Arithmetic (page 161) for a full discussion of this behavior.

To remove temporary files, you can use `on.exit` together with the `unlink` function. For example:

```
fcn.A <- function(data, file=tempfile("fcn"))
{
  on.exit(unlink(file))
  dput(data, file=file)
  #
  # additional commands
  #
}
```

The `unlink` function permanently deletes external files from inside of Spotfire S+.

Wrap-up actions specified by multiple calls to `on.exit` can be executed sequentially. Alternatively, later actions can replace earlier ones. The behavior that occurs is determined by the `add` argument to `on.exit`; by default, `add=T` and actions are executed sequentially. For example, the following function uses `on.exit` to both unlink a file and restore graphics parameters:

```
fcn.B <- function(data, file=tempfile("fcn"))
```

```
{
  on.exit(unlink(file)
  oldpar <- par()
  on.exit(par(oldpar))
  par(mfrow = c(3,4))
  #
  # make some plots and edit some data
  #
  dput(data, file=file)
  ...
}
```

If `add=F`, the new action replaces any pending wrap-up actions. For example, suppose your function performs a long, iterative computation and you want to write the last computed value to disk in case of an error. You can use `on.exit` to accomplish this as follows:

```
fcn.C <- function()
{
  for(i in 1:10000) {
      result <- i^2
      on.exit(assign("intermediate.result", result,
          where=1), add=F)
  }
  on.exit(add=F)
  return(result)
}
```

If we call this function and then interrupt the computation with ESC (Windows) or CTRL-C (UNIX), we see that the object `intermediate.result` is created. If we let the function complete, it is not the following on Windows:

```
> fcn.C()
User interrupt requested
Use traceback() to see the call stack
```

The output message is similar on UNIX:

```
> fcn.C()
Interrupt
Use traceback() to see the call stack
```

```
> intermediate.result
[1] 665856

> rm(intermediate.result)
> fcn.C()
[1] 1e+08

> intermediate.result
Problem: Object "intermediate.result" not found
```

WRITING SPECIAL FUNCTIONS

Operators

In addition to the built-in operators discussed in the section Function Names and Operators (page 153), Spotfire S+ 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 can print them only by using the `get` function:

```
> get("%*%")

function(x, y, ...)
UseMethod("%*%")
```

Here is the code for an operator that raises 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 a single number")
  if(all.equal(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)
  }
  return(m)
}
```

Once defined, this operator can be used exactly as any other infix operator:

```
> x <- matrix(c(2,1,1,1), ncol=2)
> x

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

> x %^% 3

     [,1] [,2]
[1,]   13    8
[2,]    8    5
```

You can also use this operator to find the inverse of a matrix:

```
> x %^% -1

     [,1] [,2]
[1,]    1   -1
[2,]   -1    2
```

User-defined operators have precedence equivalent to the built-in operators `%%`, `%/%`, and `%*%`. See Table 7.1 (page 154) for a complete list of built-in operators and their precedence.

Extraction and Replacement Functions

As we mention in the section Function Names and Operators (page 153), Spotfire S+ handles assignments in which the left side is a function call differently from those in which the left side is a name. An expression of the form `f(x) <- value` is evaluated as the following assignment:

```
x <- "f<-"(x, value)
```

This requires a function named `"f<-"` that corresponds to `f`. In this example, `f` is called an *extraction function*: it accepts a data object and returns either a portion of the data or some attribute of it. The function `"f<-"` is the corresponding *replacement function*: it replaces the object extracted by `f` with a user-supplied value.

For example, the `dim` function returns the `dim` attribute of a matrix, data frame, or array:

```
> x <- matrix(1:10, nrow=5)
> x

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

> dim(x)
[1] 5 2
```

The result from `dim` states that the matrix `x` has 5 rows and 2 columns. The corresponding function `"dim<-"` replaces the `dim` attribute with a user-specified value:

```
> dim(x) <- c(2,5)
> x

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

Spotfire S+ includes many replacement functions. Most notably, functions associated with subscripting often have corresponding replacements; examples include `"[<-"` and `"[[<-"`. In addition, functions associated with attribute extraction have replacement functions; `"dim<-"`, `"names<-"`, and `"class<-"` are examples. Because the names of replacement functions are not syntactic names, you must use the `get` function to examine their definitions:

```
> get("dim<-")

function(x, value)
.Internal("dim<-"(x, value), "S_replace", T, 10)
```

In general, you should define replacement functions whenever you write new extraction functions. New extraction functions are generally associated with newly-created attributes. As a simple example, suppose we want a new attribute named `doc` that holds a brief description of an object. The corresponding extraction function, `doc`, starts naturally enough with the `attr` function. A simple version of `doc` is the following one-liner:

```
doc <- function(x) { attr(x, "doc") }
```

The replacement function "doc<-" looks like:

```
"doc<-" <- function(x, value)
{
  attr(x, "doc") <- value
  return(x)
}
```

Two things are worth noting about the definition of "doc<-". First, it returns the complete, modified object and not just the modified attribute. Second, it performs no assignment; the Spotfire S+ evaluator performs the actual assignment. These characteristics are essential for writing clean replacement functions.

The following commands use the "doc<-" function to add a doc attribute to the built-in data set geyser. The attribute is then printed with the doc function:

```
# Assign geyser to your working directory.
> geyser <- geyser

> doc(geyser) <- "Waiting time between eruptions and the
Continue string: duration of the eruption for the Old
Continue string: Faithful geyser in Yellowstone."

> doc(geyser)

[1] "Waiting time between eruptions and the\nduration of
the eruption for the Old\nFaithful geyser in Yellowstone."
```

Because of the newline characters, this is not the most readable form. However, if we modify the doc function slightly to use cat instead, we obtain output that is easier to read:

```
> doc <- function(x) { cat(attr(x, "doc"), sep="\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 that you are interested in. Such functions typically use other extraction functions as their starting points. For example, the following functions use subscripting to find the elements of an input vector that have even and odd indices:

```
evens <- function(x)
{
  indices <- seq(along = x)
  return(x[indices %% 2 == 0])
}

odds <- function(x)
{
  indices <- seq(along = x)
  return(x[indices %% 2 == 1])
}
```

The following examples illustrate these functions:

```
> evens(1:10)
[1] 2 4 6 8 10

> odds(1:10)
[1] 1 3 5 7 9
```

In `evens` and `odds`, we build on the subscripting function `"["` to extract particular subsets of the input data. Thus, the subscripting replacement function `"[<-"` is the logical place to start in writing the corresponding replacements `"evens<-"` and `"odds<-"`:

```
"evens<-" <- function(x, value)
{
  indices <- seq(along = x)
  x[indices %% 2 == 0] <- value
  return(x)
}

"odds<-" <- function(x, value)
{
  indices <- seq(along = x)
  x[indices %% 2 == 1] <- value
  return(x)
}
```

The following examples illustrate replacement using these two functions:

```
> xx <- 1:10
> xx
[1]  1  2  3  4  5  6  7  8  9 10

> odds(xx) <- c(10,20,30,40,50)
> evens(xx) <- c(11,21,31,41,51)
> xx
[1] 10 11 20 21 30 31 40 41 50 51
```

As a final example of extraction and replacement, consider the problem of extracting and replacing row names in a matrix. Normally, you extract the names using the `dimnames` function and replace them using `"dimnames<-"`. However, it would be convenient to simply type `rownames(x)` and see the row names of a matrix `x`. Here is a simple function that does this:

```
rownames <- function(x)
{
  if(!is.null(dimnames(x)[[1]]))
     return(dimnames(x)[[1]])
  else
     return(character(dim(x)[1]))
}
```

If the first element of `dimnames(x)` is `NULL`, the `rownames` function returns a vector of empty character strings that has length equal to the number or rows in `x`.

The corresponding replacement function inserts new row names while preserving any existing column names:

```
"rownames<-" <- function(x, value)
{
  if(!is.null(dimnames(x)[[2]]))
     colnames <- dimnames(x)[[2]]
  else
     colnames <- NULL
  dimnames(x) <- list(value, colnames)
  return(x)
}
```

The following commands illustrate the `rownames` and `"rownames<-"` functions using the built-in data set `state.x77`:

```
> rownames(state.x77)

 [1] "Alabama"        "Alaska"         "Arizona"
 [4] "Arkansas"       "California"     "Colorado"
 [7] "Connecticut"    "Delaware"       "Florida"
[10] "Georgia"        "Hawaii"         "Idaho"
[13] "Illinois"       "Indiana"        "Iowa"
[16] "Kansas"         "Kentucky"       "Louisiana"
[19] "Maine"          "Maryland"       "Massachusetts"
[22] "Michigan"       "Minnesota"      "Mississippi"
[25] "Missouri"       "Montana"        "Nebraska"
[28] "Nevada"         "New Hampshire"  "New Jersey"
[31] "New Mexico"     "New York"       "North Carolina"
[34] "North Dakota"   "Ohio"           "Oklahoma"
[37] "Oregon"         "Pennsylvania"   "Rhode Island"
[40] "South Carolina" "South Dakota"   "Tennessee"
[43] "Texas"          "Utah"           "Vermont"
[46] "Virginia"       "Washington"     "West Virginia"
[49] "Wisconsin"      "Wyoming"

# Assign state.x77 to your working directory.
> state.x77 <- state.x77
> rownames(state.x77) <- c(LETTERS[1:25], letters[1:25])
> rownames(state.x77)

 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L"
[13] "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X"
[25] "Y" "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k"
[37] "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w"
[49] "x" "y"
```

REFERENCES

Chambers, J.M. (1998). *Programming with Data: A Guide to the S Language*. New York: Springer-Verlag.

Venables, W.N. and Ripley, B.D. (2000). *S Programming*. New York: Springer-Verlag.

DATA FRAMES 8

INTRODUCTION

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, `"logical"`, `"numeric"`, `"complex"`, `"character"`). 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 `"numeric"` values, a column of categorical values, and a column of `"logical"` 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(kyphosis),10)
> kyphosis[ sampleruns,]
     Kyphosis Age Number Start
63   present 130      4     1
 4    absent   2      5     1
39    absent   1      3     9
55   present 139     10     6
21    absent  27      4     9
46    absent  61      4     1
23   present 105      6     5
68    absent  17      4    10
18    absent  78      6    15
36    absent 112      3    16
```

A sample of 10 of the 83 observations is presented for all four variables. The variable `Kyphosis` is the outcome which indicates whether the post-operative deformity is present or absent. The row names on the left are the run numbers for the experiment. Combined in `kyphosis` are character data (the row names), categorical data (`Kyphosis`), and numeric data (`Number` and `Start`).

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 Spotfire S+. It is discussed in detail in Data Input on page 208. 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 271).

For most purposes, when you want to create or modify data frames within Spotfire S+, 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 my.matrix
 1     F  0.6089225+0.9225925840i -3.10164384  0.88012586
 2     F -2.0819531+0.0336728902i -0.55111325  0.27279513
 3     T  0.8878321+0.9771889383i -0.72223763  0.84707218
 4     F  0.7471270+0.5487224348i -0.27917610  2.00179088
 5     T  1.1005395+0.0631634402i  0.15104893 -0.68347069
 6     T  0.3485193+0.1848195572i -0.44838152 -0.47493315
 7     F  1.6454204+0.6908840677i  0.44405148  1.18727220
 8     F  1.4330907+0.0004898258i  0.04847902 -2.17772281
 9     F -0.8531461+0.9480914157i  0.14967287 -2.25494105
10     T  0.8741626+0.1823104322i -1.39863545 -3.22639704
11     F -0.2090367+0.0066157957i -0.23842813 -0.36280708
12     F  1.1757802+0.6762467814i  0.32989672 -0.86669093
13     F -0.3004673+0.3170390362i -1.68374843  0.01818504
14     T -1.4100635+0.3889551479i -1.27312064  0.35701916
16     F  0.6641272+0.2191691762i -0.60716481 -0.40695505
17     T -0.1866811+0.8029941772i -1.01767418 -1.53522281
18     F  0.8642104+0.6114265160i -0.07657414  0.23754561
19     F -0.4507343+0.6050521806i -0.38748806  0.25455890
20     F -1.8629536+0.7581159561i -1.07376663 -0.16346027
21     T -1.0725881+0.5973116844i  1.91706202  0.42669240

    my.matrix Kyphosis Age Number
 1 -0.023943300   absent  71      3
 2 -1.301475283   absent 158      3
 3 -1.396698458  present 128      4
 4  0.384949812   absent   2      5
 5 -0.639857710   absent   1      4
 6  1.134441750   absent   1      2
 7 -1.902422316   absent  61      2
 8 -0.058446250   absent  37      3
 9  0.126896172   absent 113      2
10  0.795556284  present  59      6
```

```
11  0.593684564  present  82       5
12  0.291224646   absent 148       3
13 -0.162832145   absent  18       5
14  0.248051730   absent   1       4
16 -0.957828145   absent 168       3
17  0.051553058   absent   1       3
18 -0.294367576   absent  78       6
19 -0.001231745   absent 175       5
20 -0.225155320   absent  80       5
21 -0.192293286   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
    Kyphosis 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 `kyphosis`, 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 274).

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 vectors, however, are converted to factors before being included in the data frame (assuming `options("stringsAsFactors")` is set to true). The conversion is done because Spotfire S+ assumes that character 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 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=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,reliab,mileage,type,
+ row.names=c("Acura","Audi","BMW","Chev","Ford",
+ "Mazda","MazdaMX","Nissan","Olds","Toyota"))
        price  country reliab mileage       type
  Acura 11950    Japan      5      NA      Small
   Audi 26900  Germany     NA      NA     Medium
. . .
```

Rectangular Data Functions

Rectangular data functions allow you to access all rectangular data objects in the same way. Rectangular data objects include matrices, data frames, and atomic vectors which have the form of rows (observations) and one or more columns (variables).

There are eight rectangular data functions you can use:

- `as.rectangular` converts any object to a rectangular data object (generally a data frame).
- `as.char.rect` takes a rectangular object and returns a rectangular object consisting of character strings, suitable for printing (but not formatted to fixed width).
- `is.rectangular` tests whether an object is rectangular.
- `sub` is used for subscripting.

- `numRows` and `numCols` count the number of rows and columns, respectively.
- `rowIds` and `colIds` return the row and column names, respectively.

`numRows`, `numCols`, `rowIds`, and `colIds` can also be used on the left side of assignments. For more information on any of these functions, type

```
help(function)
```

where *function* is one of the rectangular data functions listed above.

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. This section discusses three general cases:

1. Combining data frames *by column*. This is useful 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 is used 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 is useful when you have two data frames containing 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; you are simply adding another column to the existing data frame, and the natural tool for this in Spotfire S+ 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
  X1 Weight Disp. Mileage FuelType
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.mine <- cbind(cu.dimensions[match.dims,],
+ cu.specs[match.specs,], cu.summary[match.summary,],
+ row.names=common.names)
```

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

You can get statistics on individual columns by running any of the four following functions in S-PLUS:

- `colMeans`
- `colSums`
- `colVars`
- `colStdevs`

which returns the mean, sum, variance, and standard deviation, respectively, for the specified column or columns.

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 <-
data.frame(norm=rnorm(20),unif=runif(20),binom=rbinom(20,10
,0.5))
> rand.df1
          norm       unif  binom
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 <-
data.frame(norm=rnorm(20),binom=rbinom(20,10,0.5),
chisq=rchisq(20,10))
> rand.df2
        norm binom     chisq
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 `binom`; we subscript and combine the resulting data frames as follows.

```
> rbind(rand.df1[,c("norm","binom")],
+ rand.df2[,c("norm", "binom")])
         norm  binom
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`) are used only when you have complete data frames, as above. Do not use it in a loop to add one row at a time to an existing data frame (very inefficient). To build a data frame, write all the observations to a data file and use `read.table` to read it.

You can get basic statistics on individual rows by running any of the four following functions in S-PLUS:

- `rowMeans`
- `rowSums`
- `rowVars`
- `rowStdevs`

which return the mean, sum, variance, and standard deviation, respectively, for the specified row or rows.

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     Blye  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           Blye   Midwifery
3           Loren           Blye   Gardening
4           Loren           Blye  Perennials
5           Robin          Green Who_dun_it?
6           Rich           Calaway      Splus
```

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     Blye  40   40000 Washington   Midwifery
2      Loren     Blye  40   40000 Washington   Gardening
3      Loren     Blye  40   40000 Washington  Perennials
4      Lorne    Green  82 1200000 California     Bonanza
5      Robin    Green  45   25000 Washington Who_dun_it?
```

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.

Converting Data Frames

You may want to convert a S-PLUS data frame to a matrix. If so, there are three different functions which take a data frame as an argument and return a matrix whose elements correspond to the elements of the data frame:

- `as.matrix.data.frame`
- `numerical.matrix`
- `data.matrix`

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 Spotfire S+, 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 deltat 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 deltat 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 deltat 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(kyphosis, INDICES=kyphosis$Kyphosis, FUN=summary)
kyphosis$Kyphosis:absent
   Kyphosis          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
               Median: 79.00     Median:4.00    Median: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
-------------------------------------------------------
kyphosis$Kyphosis:present
   Kyphosis         Age             Number           Start
  absent: 0Min.: 15.00      Min.: 3.000     Min.: 1.000
 present:171st Qu.: 73.001st Qu.: 4.000 1st Qu.: 5.000
        Median:105.00     Median: 5.000     Median: 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(kyphosis, list(Kyphosis=kyphosis$Kyphosis,
+      Older=kyphosis$Age>105),
+      function(data)lm(Number~Start,data=data))
Kyphosis:absent
Older:FALSE
Call:
lm(formula = Number~Start, data = data)

Coefficients:
 (Intercept)       Start
    4.885736 -0.08764492
Degrees of freedom: 39 total; 37 residual
Residual standard error: 1.261852
```

```
Kyphosis:present
Older: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
Older: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$Kyphosis, function(data)
+ sapply(data,mean))
Warning messages:
1: A factor is not numeric.  Returning NA for the mean. in:
FUN(...X.sub.i....)
2: A factor is not numeric.  Returning NA for the mean. in:
FUN(...X.sub.i....)
kyphosis$Kyphosis:absent
 Kyphosis      Age Number    Start
       NA 79.89063   3.75 12.60938
-------------------------------------------------------------
----------
kyphosis$Kyphosis:present
 Kyphosis      Age   Number    Start
       NA 97.82353 5.176471 7.294118
```

```
> by(kyphosis, kyphosis$Kyphosis, function(data)
+ sapply(data,max))
Problem in Summary.factor(...X.sub.i....): A factor is not
a numeric object
Use traceback() to see the call stack
```

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] <- "Life.Exp"
> by(state.x77[,c("Murder", "Population", "Life.Exp")],
+ state.region, summary)
INDICES:Northeast
     Murder           Population        Life.Exp
Min.    : 2.400   Min.    :  472    Min.    :70.39
1st Qu.: 3.100   1st Qu.:  931    1st Qu.:70.55
Median : 3.300   Median : 3100    Median :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          Population        Life.Exp
Min.    : 6.20   Min.    :  579    Min.    :67.96
1st Qu.: 9.25   1st Qu.: 2622    1st Qu.:68.98
Median :10.85   Median : 3710    Median :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
```

ADDING NEW CLASSES OF VARIABLES TO DATA FRAMES

The manner in which objects of a particular data type are included in a data frame is determined by that type's method for the generic function `as.data.frame`. The default method for this generic function uses the `data.class` function to determine an object's *type*. Thus, even data types without formal `class` attributes, such as vectors, or character vectors, can have specific methods. The behavior for most built-in types is derived from one of the six basic cases shown in the table below.

Table 8.1: *Rules for combining objects into data frames.*

Data Types	Sub-types	Rules

`vector`	`numeric`
	`complex`
	`factor`
	`ordered`
	`rts`
	`its`
	`cts`
		1. contribute a single variable as is

`character`	`character`
	`logical`
	`category`
		1. converted to a factor data type
		2. contribute a single variable

`matrix`	`matrix`	1. each column creates a separate variable. 2. column names used for variable names
`list`	`list`	1. each component creates one or more separate variables 2. variable names assigned as appropriate for individual components (column names for matrices, etc.)

Table 8.1: *Rules for combining objects into data frames. (Continued)*

Data Types	Sub-types	Rules
`model.matrix`	`model.matrix`	1. object becomes a single variable in result
`data.frame`	`data.frame` `design`	1. each variable becomes a variable in result design. 2. variable names used for variable names

As you add new classes, you can ensure that they are properly behaved in data frames by defining your own `as.data.frame` method for each new class. In most cases, you can use one of the six paradigm cases, either as is or with slight modifications. For example, the character method is a straightforward modification of the vector method:

```
> 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] "Price"  "Country"  "Reliab"  "Mileage"  "Type"

$row.names:
 [1]   "AcuraIntegra4"         "Audi1005"        "BMW325i6"
 [4]     "ChevLumina4"     "FordFestiva4"     "Mazda929V6"
 [7]  "MazdaMX-5Miata"   "Nissan300ZXV6"    "OldsCalais4"
[10] "ToyotaCressida6"

$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] "Brazil"    "England"    "France"     "Germany"
[5] "Japan"     "Japan/USA"  "Korea"      "Mexico"
[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 80). 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 8.2: *Attributes of Data Frames.*

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

IMPORTING AND EXPORTING

9

SUPPORTED FILE TYPES FOR IMPORTING AND EXPORTING

Table 9.1 lists all the supported file formats for importing and exporting data. Note that Spotfire S+ both imports from and exports to all the listed types with two exceptions: SigmaPlot (**.jnb**) files are import only and HTML (**.htm***) tables are export only.

Table 9.1: *Supported file types for importing and exporting data.*

Format	Type	Default Extension	Notes
ASCII File	`"ASCII"`	**.csv** **.asc**, **.csv**, **.txt**, **.prn** **.asc**, **.txt**, **.prn**	Comma delimited. Delimited. Whitespace delimited; space delimited; tab delimited; user-defined delimiter.
dBASE File	`"DBASE"`	**.dbf**	II, II+, III, IV files.
DIRECT-DB2	DB2 database connection.		No file argument should be specified.
DIRECT-ORACLE	Oracle database connection.		No file argument should be specified.
DIRECT-SQL	Microsoft SQL Server database connection.		No file argument should be specified. Available on Windows® only.
DIRECT-SYBASE	Sybase database connection.		No file argument should be specified
Epi Info File	`"EPI"`	**.rec**	

Table 9.1: *Supported file types for importing and exporting data. (Continued)*

Format	**Type**	**Default Extension**	**Notes**
Fixed Format ASCII File	`"FASCII"`	**.fix**, **.fsc** (Windows)	
FoxPro File	`"FOXPRO"`	**.dbf** (Windows)	Uses same import filter as dBASE files.
Gauss Data File	`"GAUSS"`, `"GAUSS96"`	**.dat**	Automatically reads the related **DHT** file, if any, as `GAUSS 89`. If no **DHT** file is found, reads the **.DAT** file as `GAUSS96`.
HTML Table	`"HTML"`	**.htm***	Export only.
Lotus 1-2-3 Worksheet	`"LOTUS"`	**.wk***, **.wr***	
MATLAB Matrix	`"MATLAB"` `“MATLAB7”` `(export only)`	**.mat**	File must contain a single matrix. Spotfire S+ recognizes the file's platform of origin on import. On export, specify `type="MATLAB"` to create a pre-MATLAB 7 version file; otherwise, specify `type="MATLAB7"` to export the MATLAB 7 file format.
Minitab Workbook	`"MINITAB"`	**.mtw**	Versions 8 through 12.
Microsoft Access File	`"ACCESS"`	**.mdb** (Windows)	
Microsoft Excel Worksheet	`"EXCEL"` `"EXCELX"`	**.xl?** **.xlsx**	Versions 2.1 through 2010. Note that `"EXCELX"` and the new file extension, `".xlsx"` are for files imported from or exported to Excel 2007 and 2010.

Table 9.1: *Supported file types for importing and exporting data. (Continued)*

Format	**Type**	**Default Extension**	**Notes**
Microsoft SQL Server	`"MS-SQL"`	**.sql**	
ODBC Database	`"ODBC"`	Not applicable	For Oracle (**.ora**) and SYBASE (**.syb**) databases (Windows only).
Oracle	`"ORACLE"`	**.ora**	Oracle database connection. No `file` argument should be specified (UNIX®).
Paradox Data File	`"PARADOX"`	**.db** (Windows)	
QuattroPro Worksheet	`"QUATTRO"`	**.wq?, .wb?**	
S-PLUS File	`"SPLUS"`	**.sdd**	Windows, DEC UNIX. Uses `data.restore()` to import file.

STATA	`“STATA”`
	
	`“STATASE”` `(export only)`
		.dta	Portable across platforms (UNIX, Windows, and Mac). Can import STATA files and export STATA or STATASE.
			
			When exporting a STATA dataset, you are limited to 2,047 characters. For larger STATA datasets (up to 32,767 variables), specify `type="STATASE"` .

Table 9.1: *Supported file types for importing and exporting data. (Continued)*

Format	**Type**	**Default Extension**	**Notes**
SAS File	`"SAS"`, `"SASV6"`	**.sd2**	SAS version 6 files, Windows.
	`"SAS1"`, `"SAS6UX32"`	**.ssd01**	SAS version 6 files, HP, IBM, Sun UNIX.
	`"SAS4"`, `"SAS6UX64"`	**.ssd04**	SAS version 6 files, Digital UNIX.
	`"SAS7"`	**.sas7bdat**, **.sd7**	SAS version 7 or 8 files, current platform.
	`"SAS7WIN"`	**.sas7bdat**, **.sd7**	SAS version 7 or later data files (Windows).
	`"SAS7UX32"`	**.sas7bdat**, **.sd7**	SAS version 7 or later data files, Solaris (SPARC), HP-UX, IBM AIX.
	`"SAS7UX64"`	**.sas7bdat**, **.sd7**	SAS version 7 or later data files, Digital/Compaq UNIX.
SAS Transport File	`"SAS_TPT"`	**.xpt**, **.tpt**	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.
SigmaPlot File	`"SIGMAPLOT"`	**.jnb**	Import only. Available on Windows only.
SPSS Data File	`"SPSS"`	**.sav**	OS/2; Windows; HP, IBM, Sun, DEC UNIX.
SPSS Portable File	`"SPSSP"`	**.por**	

Table 9.1: *Supported file types for importing and exporting data. (Continued)*

Format	**Type**	**Default Extension**	**Notes**
Stata Data File	`"STATA"`	**.dta**	Versions 2.0 and higher.
SYBASE Server	`"SYBASE"`	**.syb**	SSYBASE database connection. No `file` argument should be specified.
SYSTAT File	`"SYSTAT"`	**.syd**, **.sys**	Double- or single-precision **.sys** files.

IMPORTING DATA

Using the importData Function

The principal tool for importing data is the importData function, which can be invoked from either the Spotfire S+ prompt or the **File ▶ Import Data** menu option.

In most cases, all you need to do to import a data file is to call importData with the name of the file to be imported as the only argument. As long as the specified file has one of the default extensions listed in Table 9.1, you need not specify a type nor, in most cases, any other information.

For example, suppose you have a SAS data file named **rain.sd2** in your start-up folder (Windows) or directory (UNIX). You can read this file into Spotfire S+ using importData as follows:

```
> myRain <- importData("rain.sd2")
```

If you have trouble reading the data, most likely you just need to supply additional arguments to importData to specify extra information required by the data importer to read the data correctly. Table 9.2 lists the arguments to the importData function.

Table 9.2: *Arguments to* importData.

Argument	Required or Optional	Description
file	Required (except for database reads)	A character string specifying the name of the file and directory path.
type	Optional	A character string specifying the file type of the file to be imported. See the "Type" column of Table 9.1 for a list of possible values.
keep	Optional	A character vector of variable names, or a numeric vector of column numbers, specifying which variables are to be imported. Only one of keep or drop may be specified.

Table 9.2: *Arguments to `importData`. (Continued)*

Argument	Required or Optional	Description
`drop`	Optional	A character vector of variable names, or a numeric vector of column numbers, specifying which variables are *not* to be imported. Only one of `keep` or `drop` may be specified.
`colNames`	Optional	A character vector of names to use for the imported columns. See the `importData` help file for more detailed information on this argument.
`rowNamesCol`	Optional	An integer specifying the column that contains the row names. If specified, the column of row names is dropped from the resulting data frame.
`filter`	Optional	A character string containing a logical expression for selecting the rows to be imported. For details, see Filter Expressions on page 289.
`format`	Optional	A single character string specifying the format for each field when importing from a formatted ASCII (FASCII) text file. For details, see Notes on Importing Files of Certain Types on page 291.
`delimiter`	Optional	A character string specifying the delimiter to use. This argument is used only when importing ASCII text files.
`startCol`	Optional	An integer specifying the starting column in the source. For example, if you specify `5`, Spotfire S+ begins reading the data at column 5.
`endCol`	Optional	An integer specifying the ending column in the source. The default of `-1` means to read to the last column.

Table 9.2: *Arguments to* `importData`. *(Continued)*

Argument	Required or Optional	Description
`startRow`	Optional	An integer specifying the starting row in the source. For example, if you specify `10`, Spotfire S+ begins reading the data at row 10.
`endRow`	Optional	An integer specifying the ending row in the source. The default of `-1` means to read to the last row.
`pageNumber`	Optional	The page number of the spreadsheet (used only for spreadsheets).
`colNameRow`	Optional	An integer specifying the row that contains the column names (used only for spreadsheets). If you do not specify a row, Spotfire S+ attempts to locate column names in the first row of the file. Specify `0` to tell Spotfire S+ not to search for a column names row. In a delimited ASCII file, the column names row must come before the first data row (`startRow`) to be read.
`server`	Optional	When importing from a relational database, a character string specifying the database server.
`user`	Optional	When importing from a relational database, a character string specifying the user name.
`password`	Optional	A character string specifying the password for the database user.
`database`	Optional	A character string specifying the name of the database to use when importing from a relational database. This should be set to `""` if `type="ORACLE"`.

Table 9.2: *Arguments to* `importData`. *(Continued)*

Argument	Required or Optional	Description
`table`	Optional	A character string specifying the name of the table in `database` to import. When you import from a database, you cannot specify table in conjunction with the sqlQuery argument. You can use `table` to specify which sheet or page to retrieve from a worksheet file type, such as Excel. To specify which sheet to import, set `table` to the sheet name. The sheet name must match exactly, including case. If you do not know the name, use the functon `contentsData` to retrieve the names of all sheets in the file, or specify the sheet number using the `pageNumber` argument. If you omit `table`, then `pageNumber` is used. The `table` argument value is ignored if `pageNumber` is set (unless `pageNumber` is invalid, in which case `table` is used). You can use `table` to specify which dataset to retrieve from a SAS Transport file. To specify which dataset to import, set `table` to the dataset name. The dataset name must match exactly, including the case. If you do not know the name, use the `pageNumber` argument and specify the number of the dataset in the file. If you omit the `table` argument, the first dataset in the file is imported.
`stringsAsFactors`	Optional	A logical flag. If `TRUE`, strings are converted to factors when imported.
`sortFactorLevels`	Optional	If `sortFactorLevels=TRUE`, the levels for all factors created from character strings are sorted. Otherwise, the order of the levels is not specified. In previous versions of Spotfire S+, there were situations where importing with `sortFactorLevels=FALSE` was significantly faster, but this is no longer true. This argument is not supported when reading a big data object (`bigdata=T`).

Table 9.2: *Arguments to `importData`. (Continued)*

Argument	Required or Optional	Description
`valueLabelAsNumber`	Optional	A logical flag. If `TRUE`, SAS and SPSS variables (numeric or character) with labels are imported as data values. If `FALSE`, the variables are imported as value labels.
`centuryCutoff`	Optional	A numeric value specifying the origin for two-digit dates. Dates with two-digit years are assigned to the 100-year span beginning with this value. The default value of 1930 means that the date 6/15/30 will be read as June 15, 1930 and 12/29/29 will be read as December 29, 2029. This argument is used only when importing from an ASCII file.
`separateDelimiters`	Optional	A logical flag. If `TRUE`, the separator is strictly a single character; otherwise, repeated consecutive separator characters are treated as one separator.
`odbcConnection`	Required if `type="ODBC"`	An encrypted character string containing the ODBC connection string. *Not currently supported on UNIX platforms.*
`odbcSqlQuery`	Optional	Contains an optional SQL query. If no query is specified, the first table of the data source is used. Meaningful only if `type="ODBC"`. *Not currently supported on UNIX platforms.*
`readAsTable`	Optional	A logical flag. If `TRUE`, Spotfire S+ reads the entire file as a single table.
`colNamesUpperCase`	Optional	A logical flag. If `TRUE`, column names are imported in all uppercase.
`time.in.format`	Optional	A character string specifying the format to use to interpret date/time data when importing from an ASCII or FASCII text file.

Table 9.2: *Arguments to* `importData`. *(Continued)*

Argument	Required or Optional	Description
`decimal.point`	Optional	A single character specifying the decimal point character for ASCII data files. By default, this is the period (.).
`thousands.separator`	Optional	A single character specifying the thousands separator character for ASCII data files. By default, this is the comma (,).
`time.zone`	Optional	A string naming the time zone any dates in the input are assumed to be in. Currently, time zone information in the data file is ignored. This argument is not supported when reading a big data object (`bigdata=T`).
`use.locale`	Optional	A logical value. If `use.locale=TRUE`, the default values of `decimal.point` and `thousands.separator` come from the current locale set by `Sys.setlocale`, and the default value of `time.zone` is `options()$time.zone`. Otherwise, the default values are as described above.
`sqlReturnData`	Optional	A logical value. If `sqlReturnData=TRUE` (the default), any SQL query expression is evaluated and the resulting data is returned. If `sqlReturnData=FALSE`, the SQL query is executed for effect only and `NULL` is returned. See the `importData` help file for details.
`scanLines`	Optional	An integer giving the number of lines that will be scanned from an ASCII input file before performing the import to determine the column name and types and widths. Specifying a negative value such as `scanLines=-1` means to scan the entire file, which may take a long time for large files, but is the safest option. See the `importData` help file for details.

Table 9.2: *Arguments to `importData`. (Continued)*

Argument	Required or Optional	Description
`maxLineWidth`	Optional	An integer giving the maximum line width expected when reading ASCII text files. If a line is read that is longer than this value, an error is signaled. The default of 0, or any number less than 32768 is treated as 32768.
`na.string`	Optional	A character string or numeric value that will be read as a missing value when reading an ASCII text file or an Excel file. No matter what value is specified for this argument, an empty character string or numeric value will always be read as a missing value.
`colTypes`	Optional	A character vector of column types to use for the imported columns. This can contain values from `"numeric"`, `"character"`, `"factor"`, and `"timeDate"`.
`sasFormats`	Optional	Specifies the SAS formats file. See the NOTE section in the `importData` help file for more detail.
`bigdata`	Optional	A logical value. If `TRUE`, the data is read into a big data object or type `bdFrame`. Otherwise, it is read into a `data.frame` object. This argument can be used only if the bigdata library section has been loaded.

Filter Expressions

The `filter` argument to `importData` 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 Spotfire S+ 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 `importData` or `exportData`.

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, as well as the relational operators listed in Table 9.3.

Table 9.3: *Relational operators.*

Operator	Description
`==`	Equal to
`!=`	Not equal to

Table 9.3: *Relational operators. (Continued)*

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
&	And
|	Or
!	Not

Examples

Examples of selection conditions given by filter expressions are:

```
"sex = 1 & age < 50"
"(income + benefits) / famsize < 4500"
"income1 >=20000 | 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 have special characters or embedded blanks:

```
"'Disp.' > 300"
```

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 2s 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 of Certain Types

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 to skip over one or more columns when importing.

Use the `format` argument to `importData` to specify the data types of the imported columns. (Note that field-width specifications are irrelevant for ASCII files and are ignored.) For each column, you need to specify a percent sign (`%`) and then the data type. Dates may be imported automatically as numbers. After importing, you can change the column format type to a dates format.

Here is an example format string:

```
%s, %f, %*, %f
```

The `s` denotes a string data type, the `f` denotes a float data type (actually, `numeric`), and the asterisk (`*`) denotes a "skipped" column. These are the only allowable format types.

If you do not specify the data type of each column, Spotfire S+ 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.

Spotfire S+ auto-detects the file delimiter from a preset list that includes commas, spaces, and tabs. All cells must be separated by the same delimiter (that is, each file must be comma-separated, space-separated, or tab-separated.) 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 with a missing value. Incomplete rows are also filled with missing values.

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 Spotfire S+ 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 a FASCII data file).

When importing a FASCII file, you need to specify a value for the `colNames` argument to `importData`. 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, `Apples, Oranges, Pears`). You can use an asterisk (`*`) to denote a missing name (for example, `Apples, *, Pears`).

When importing a FASCII file, you also need to specify the data types and field widths of the imported columns by entering a value for the `format` argument to `importData`. For each column, you need to specify a percent 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 want to import only some of the rows, specify a starting and ending row.

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

Microsoft Excel files

If your Excel 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 `colNameRow` 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 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 `colNameRow` 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.

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

dBASE files

Spotfire S+ 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.

Data from ODBC data sources (Windows only)

To access a database on a remote server, Spotfire S+ must establish a communication link to the server across the network. The information required to create this link is contained in an ODBC *connection string*. This string consists of one or more attributes that specify how a driver connects to a data source. An attribute identifies a specific piece of information that the driver needs to know before it can make the appropriate data source connection. Each driver may have a different set of attributes, but the connection string is always of the form:

```
DSN=dataSourceName [;SERVER=value] [;PWD=value]
  [;UID= value] [;<Attribute>=<value>]
```

You must specify the data source name if you do not specify the user ID, password, server, and driver attributes. However, all other attributes are optional. If you do not specify an attribute, that attribute defaults to the value specified in the relevant DSN tab of the ODBC Data Source Administrator.

Note

For some drivers, attribute values are case-sensitive.

For example, a connection string that connects to the `Employees` data source using the `hr.db` server and user `joesmith`'s account information would be:

```
"DSN=Employees;UID=joesmith;PWD=secret;SERVER=hr.db"
```

The Spotfire S+ GUI encrypts ODBC connection strings to protect sensitive information such as user IDs and passwords. To connect to your database from the command line with an encrypted connection string, first establish connectivity from the GUI and then examine your history log by choosing **Windows** ▶ **History** ▶ **Display** from the main menu or clicking the **History Log** button on the **Standard** toolbar. Simply copy the encrypted connection string into your script or to the **Commands** window.

Note

ODBC import and export facilities do not support `"nchar"` or `"nvarchar"` data types. The `"varchar"` type is supported.

To import data from a database via ODBC, use the standard `importData` function with the `type=ODBC` argument. Three additional parameters control the call to the ODBC interface:

- `file` supplies the name of the data source;
- `odbcConnection` supplies the ODBC connection string;
- `odbcSqlQuery` supplies an optional SQL query. For example, this query would specify the table you want to import. If no query is specified, the first table of the data source is used.

For example, this command creates a new data frame called `myDataSet` and fills it with the contents of `Table23` from data source `testSQLServer`:

```
> myDataSet <-importData(
      file = "testSQLServer",
      type = "ODBC",
      odbcConnection =
      "DSN=testSQLServer;UID=joesmith;PWD=secret; APP=S-
      PLUS;WSID=joesComputer;DATABASE=testdba",
      odbcSqlQuery="Select * from testdba.dbo.Table23"
)
```

You can use the `filter` argument in the `importData` function to filter data, as described on page 289.

To export data from Spotfire S+ via ODBC, use the standard `exportData` function with the `type=ODBC` argument. Four additional parameters control the call to the ODBC interface:

- `data` supplies the data frame to be exported;
- `file` supplies the name of the data source;

- `odbcConnection` supplies the ODBC connection string;
- `odbcTable` supplies the name of the table to be created.

For example, this command exports the data frame `myDataSet` to `Table23` of data source `testSQLServer`:

```
exportData(data="myDataSet", file="testSQLServer",
  type="ODBC", odbcConnection =
      "DSN=testSQLServer;UID=joesmith;PWD=secret; APP=S-
      PLUS;WSID=joesComputer;DATABASE=testdba",
      odbcSqlQuery="Select * from testdba.dbo.Table23"
)
```

Beware that if you export data to an existing table name, it is possible to change the schema for that table. This is because Spotfire S+ essentially replaces the existing tables with a new table containing the exported data. Also note that it is not possible to append data to a table. If you wish to append data to an existing table, export the data to a dummy table and then use SQL commands on the database side to join the two tables.

A new function since S-PLUS 6.0 is `executeSql`, which sends arbitrary SQL statements to a database via ODBC. The function has the following form:

```
executeSql(odbcConnection = character(0), odbcSqlQuery =
  character(0), returnData)
```

where

`odbcConnection`	is the connection string to the database
`odbcSqlQuery`	is the statement passed to the database
`returnData`	is the flag to return the data (default=F)

The following is an example of adding a record to an existing table:

```
executeSql("DSN=mydatabase","INSERT into mytable values
('Hello')")
```

Note that if `returnData` is set to `T`, the SQL will be evaluated twice.

Data from enterprise databases (UNIX only)

The `importData` function supports importing data from Oracle and SYBASE databases by making Spotfire S+ a client that connects to the databases. The database must be properly configured for network client access, and appropriate environment variables must be set for the import to work.

The environment variables needed for Oracle are shown in Table 9.4.

Table 9.4: *Environment variables for Oracle.*

Variable	Value	Example
ORACLE_HOME	The location where ORACLE was installed	**/opt1/oracle7**
LD_LIBRARY_PATH	Need to include **$ORACLE_HOME/lib**	**/opt1/oracle7/lib**

For SYBASE, you need to have the CT-library installed. The environment variable needed for SYBASE is shown in Table 9.5.

Table 9.5: *Environment variable for SYBASE.*

Variable	Value	Example
LD_LIBRARY_PATH	Need to include the **lib** directory where CT-library was installed	**/homes/sybase/lib**

The arguments to `importData` that are required when importing from these databases are listed in Table 9.7.

Table 9.6: *Required UNIX arguments for importing data from enterprise databases.*

Argument	Description
`type`	A character string specifying either `"oracle"` or `"sybase"` as the database type.
`server`	The name of the database server. This is site-specific.
`user`	The name of the user who is allowed to connect to the database.

Table 9.6: *Required UNIX arguments for importing data from enterprise databases.*

Argument	Description
`password`	The password for `user` to connect to the database.
`database`	The name of the database to import from. For Oracle, this should be the empty string (`""`).
`table`	The table in `database` to import.

Other Data Import Functions

While `importData` is the recommended method for reading data files into Spotfire S+, there are several other functions that you can use to read ASCII data. These functions are commonly used by other functions in Spotfire S+ so it is a good idea to familiarize yourself with them.

The `scan` Function

The `scan` function, which can read either from 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
```

Spotfire S+ 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 file 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 Spotfire S+, 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 Spotfire S+ 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.

The `read.table` Function

Data frames in Spotfire S+ 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 Spotfire S+ using the `read.table` function. For example, consider a data file named `auto.dat` that contains the records listed below.

```
       Model      Price   Country Reliab Mileage Type
AcuraIntegra4     11950   Japan   5      NA      Small
Audi1005          26900   Germany NA     NA      Medium
BMW325i6          24650   Germany 94     NA      Compact
ChevLumina4       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
OldsCalais4        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
  AcuraIntegra4 11950   Japan      5      NA   Small
       Audi1005 26900 Germany     NA      NA  Medium
       BMW325i6 24650 Germany     94      NA Compact
    ChevLumina4 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
    OldsCalais4  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 Spotfire S+ from the users of your functions.

USING DIRECT DATABASE DRIVERS

Note

As of Spotfire S+ version 8.2, native database drivers are deprecated. In lieu of these drivers, you should use JDBC/ODBC drivers for all supported database vendors.

Spotfire S+ includes access to the following databases via direct database drivers:

- Microsoft SQL Server® (Windows® only).
- IBM DB2® (Windows, Solaris®32 and 64, Linux®32 and 64, Compaq Tru64®, HP®, AIX)®.
- Sybase® (Windows, Solaris 32 and 64, Linux 32, HP, AIX).
- Oracle® (Windows, Solaris 32 and 64, Linux 32 and 64, HP, AIX).

We strongly recommend using ODBC. and JDBC drivers.

You can use the `importData` and `exportData` commands to access the above databases using the new direct drivers.

Exporting to databases now allows existing tables to be either replaced or appended to.

In addition, in Spotfire S+ for Windows, you can use the **Import From Database** and **Export to Database** dialogs to access these databases as data sources, just like accessing ODBC data sources. The previous versions of Spotfire S+ had menu items such as **Import Data ▶ From ODBC Connection** and **Export Data ▶ To ODBC Connection**, and these have been replaced with **Import From Database** and **Export to Database**, respectively. The dialogs presented now allow you to select either ODBC or direct database sources from the same list.

Direct database access is accomplished by using driver components which must be separately installed on the system that is running Spotfire S+. These driver components are provided by the database vendor and usually consist of components that can be called directly by Spotfire S+ to send and receive database information in the native format that the database supports. In many cases, direct database drivers provide faster connectivity and data transfer than ODBC because there are fewer layers of data translation and interpretation of the request than with ODBC.

SPOTFIRE S+ COMMANDS FOR IMPORTING AND EXPORTING

Before you can use the `importData` or `exportData` commands to access database via the direct drivers, you must install database client software on your system. Please refer to the sections below on installing database clients for your system type.

There are four new database type keywords that can be used in the type parameter of the **importData** or **exportData** commands:

- `DIRECT-DB2`
- `DIRECT-ORACLE`
- `DIRECT-SQL`
- `DIRECT-SYBASE`

As an example, consider the following Spotfire S+ commands to send data to a Sybase database:

```
mydata <- data.frame(COL1=c(1.2,1.3,1.51,2.1,3.9),
     COL2=c("a", "b", "c", "d", "e"),
     COL3=timeDate(c("1/1/2003", "2/1/2003",
           "3/15/2005", "10/24/2003", "11/11/2004"),
           format="%02m/%02d/%Y %02H:%02M:%02S.%03N"))

exportData(mydata, type="DIRECT-SYBASE",
     user="testqa", password="testqa",
     server="qaimage.tibco.com", database="testdb",
     table="testDirectSybase", appendToTable=F)
```

In this example, the data frame `mydata` is exported to the table `testDirectSybase` into the database `testdb` on the server `qaimage.tibco.com`. The database client software validates the user and password parameter values prior to exporting the data, and if they are incorrect, an error is reported in Spotfire S+. The server name you specify here should be the one you specified during installation of the Sybase client software. See Step 6 in the Windows Sybase client installation instructions below for further information on this.

Notice that the string `DIRECT-SYBASE` was used in the `type` parameter of the `exportData` command to specify connection to a Sybase database. Also note that the user name, password, server name, database and table name are specified. For each of the four databases supported by direct drivers in Spotfire S+, slightly different combinations of these parameters must be specified. See the table below for a list of the differences.

Table 9.7: *Table of parameters required for various direct database types in* `importData` *and* `exportData`.

Database type	**Required parameters for** `importData and exportData`	**Comments**
`DIRECT-DB2`	`user, password, database, table`	Server parameter should *not* be specified for DB2.
`DIRECT-ORACLE`	`user, password, server, table`	Database parameter should not be specified for Oracle. Server parameter should be the **network service name** you specified when installing the Oracle client software. See Step 5. in the section Oracle Client.
`DIRECT-SQL`	`user, password, server, database, table`	Server parameter should be the server name you specified in the **SQL Server Enterprise Manager** program. See Step 11. in the section SQL Server 2000 Client.

Table 9.7: *Table of parameters required for various direct database types in `importData` and `exportData`.*

Database type	**Required parameters for** `importData and exportData`	**Comments**
`DIRECT-SYBASE`	`user, password, server, database, table`	Server parameter should be the server name you specified in the Sybase installation. See Step 6. in the section Sybase Client.

Notice the existence of the `appendToTable` parameter, which controls whether or not to append the data you are exporting to the specified table. If this parameter is false, the data overwrites the table specified; if true, the data is appended.

An error occurs if the data types of the data you are sending and those already present in the table do not match. For example, if you export strings to a table that currently has columns of numeric values, you receive an error and the export fails.

Here is an example of using the `importData` command to read data into Spotfire S+ from an Oracle database table via the direct drivers:

```
mynewdata <- importData(type="direct-oracle",
                user="testqa", password="testqa",
                server="ORACLE.TESTDB",
                table="testDirectOracle")
```

In this example, the table `testDirectOracle` is used. Since no specific SQL query is specified (normally specified with the `sqlQuery` parameter), all data from the table is imported. The database client software validates the user and password parameter values prior to importing the data and if they are incorrect, an error is reported in Spotfire S+. The server name you specify here should be the one you specified as the network service name during installation of the Oracle client software. See Step 5. in the section Oracle Client installation instructions below for further information on this.

For more information on using the `importData` and `exportData` commands as well as additional information on using the `sqlQuery` parameter in `importData`, please see the online help.

DIALOGS FOR IMPORTING AND EXPORTING

In Spotfire S+ for Windows, you can use the **Import From Database** and **Export to Database** dialogs to access databases as data sources, just like accessing ODBC data sources. The previous versions of Spotfire S+ had menu items such as **Import Data ▶ From ODBC Connection** and **Export Data ▶ To ODBC Connection**, and these have been replaced with **Import From Database** and **Export to Database**, respectively. The dialogs presented now allow you to select either ODBC or direct database sources from the same list.

Import From Database

You can use the Spotfire S+ **Import From Database** dialog to import table data from direct database sources available on your system. Only those sources for which you have installed database clients are supported. For more information on installing database client software, see the section Installing and Configur-ing Database Clients on UNIX and the section Installing and Configur-ing Database Clients on Windows later in this document.

Spotfire S+ lists all data sources it supports in the **Import From Database** dialog. However, only those sources which have database clients installed on your system work. The others report errors until you install and configure the appropriate database clients.

When you first run Spotfire S+, four direct data sources are listed in the **Import From Database** dialog, one for each type supported on Windows:

- Direct DB2
- Direct Oracle
- Direct SQL Server
- Direct Sybase

You can modify these data sources to configure them for your particular database setup, such as setting the correct server, database, username, password, and other information appropriate for how the database is set up on your network.

You can also add new direct sources (based on one of the four types listed above) to the list of data sources. This allows you to have one or more data sources for the same database type, thus enabling you to specify different database names or usernames and passwords to access other sets of tables on the same database server.

You can also remove direct data sources from the **Data Source** list.

To import data from a direct database source:

1. From the **File** menu, select **Import Data ► From Database**.
2. The **Import From Database** dialog appears, as shown in Figure 9.1.

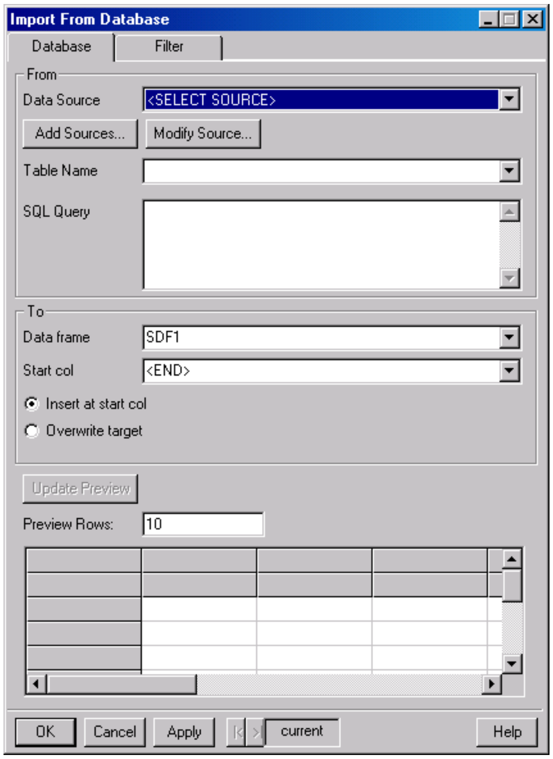


Figure 9.1: *The* ***Import From Database*** *dialog.*

3. Select a direct data source from the **Data Source** list.

   You can customize these or create new direct sources by clicking the **Add Sources** button. See below for further information on creating direct data sources.

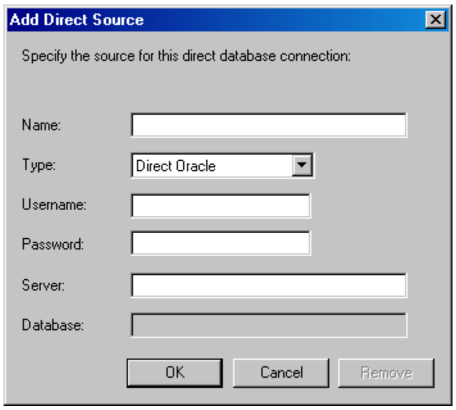
   Choose one of these or select another one you have created.

   If you have not completely configured the source, the **Modify Data Source** dialog appears. Fill in all the fields with valid information for the data source chosen to continue.

4. Once a direct data source has been selected, the **Tables** list changes to contain all the tables in that source. Select a table from the list.
5. Specify any other options, including a valid SQL query in the **SQL Query** field. If you leave the **SQL Query** field blank, a default query of all columns and rows from the selected table is performed.
6. Click the **OK** button to start the import.

You can add new direct data sources that are based on one of the four supplied direct database types. You can add as many data sources as you wish. To add a direct data source, do the following:

1. From the **File** menu, select **Import Data ▶ From Database**.
2. Click the **Add Sources** button, and from the context menu, select **Add Direct Source**.
3. The **Add Direct Source** dialog appears (Figure 9.2).


Add Direct Source

Specify the source for this direct database connection:

Name:

Type: Direct Oracle

Username:

Password:

Server:

Database:

Figure 9.2: *The* ***Add Direct Source*** *dialog.*

4. Enter the name for the new direct data source in the **Name** field. The name you enter is used to display this data source in the list of data sources in the **Import From Database** and **Export To Database** dialogs. Choose a name that is different from other entries in the data sources list.
5. Select the type of database from the drop list of database choices in the **Type** field.

6. Depending on the type you select, specify the username, password, server and database name using the fields provided. Some database types do not require a server or database name, and so those fields may be unavailable.

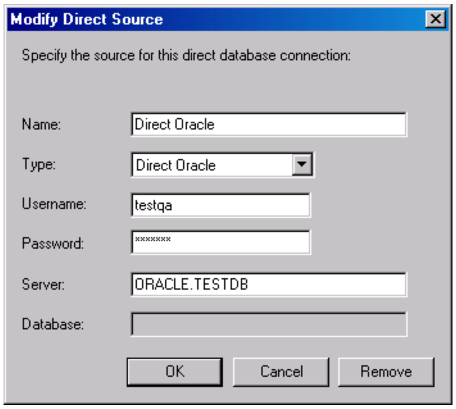
7. Click **OK** to add the source to the data sources list. When you add a source, it is also selected as the current source to import from.

You can modify each of these with the appropriate information for your database configurations. To modify a direct data source:

1. From the **File** menu, select **Import Data ▶ From Database**.

2. Select the direct data source from the **Data Sources** list you want to modify.

3. If the data source has invalid or incomplete information or you are using the data source for the first time, the **Modify Data Source** dialog appears, as shown in Figure 9.3.



Modify Direct Source

Specify the source for this direct database connection:

Name: Direct Oracle

Type: Direct Oracle

Username: testqa

Password: *******

Server: ORACLE.TESTDB

Database:

Figure 9.3: *The **Modify Direct Source** dialog.*

4. If the **Modify Data Source** dialog does not appear, it indicates that the information for this data source is valid. Click the **Modify Source** button below the list to display the **Modify Data Source** dialog.

5. In the **Modify Data Source** dialog, specify your user name and password along with the server name and database name for the source as appropriate for the database type. See Table 9.7 in the section Spotfire S+ Commands for Importing and

Exporting for help identifying which fields need to be filled out for a given database type. The fields not required for a given type are greyed out in the dialog.

6. You can also change the data source name shown in the **Name** field and the database type shown in the **Type** drop list. Changing the data source name changes how it is listed in the **Data Sources** list in the dialog for both the **Import** and **Export** dialogs. Changing the database type changes which fields are available and may require you to specify different information depending on the database type chosen. If you change the database type, it is a good idea to change the name to identify it as a different data source in the list.
7. Click the **OK** button to accept your changes.

You can also remove direct data sources. Be careful using this dialog, as you can remove the four direct data sources that are provided with Spotfire S+. If you do, you can add them again following the procedures above to add a data source. To remove a data source:

1. From the **File** menu, select **Import Data ▶ From Database**.
2. Select the direct data source from the **Data Sources** list you want to modify.
3. Click the **Modify Source** button. The **Modify Direct Source** dialog appears.
4. Click the **Remove** button in this dialog to remove the data source.

Export to Database

You can use the Spotfire S+ **Export to Database** dialog to export data frame objects from Spotfire S+ to direct database sources available on your system. Only those sources that you have installed database clients for are supported. For more information on installing database client software, see the Installing and Configuring sections later in this document.

Spotfire S+ lists all data sources it supports in the **Export to Database** dialog. However, only those sources which have database clients installed on your system work. The others sources report errors until you install and configure the appropriate database clients.

When you first run Spotfire S+, four direct data sources are listed in the **Export To Database** dialog, one for each type supported on Windows. These are listed as follows:

- Direct DB2
- Direct Oracle
- Direct SQL Server
- Direct Sybase

You can modify these data sources to configure them for your particular database setup, such as setting the correct server, database, username, password, and other information appropriate for how the database is setup on your network.

You can also add new direct sources (based on one of the four types listed above) to the list of data sources. This allows you to have one or more data sources for the same database type, thus enabling you to specify different database names or usernames and passwords to access other sets of tables on the same database server. You can also remove direct data sources from the data sources list.

To export data to a direct database source:

1. From the **File** menu, select the **Export Data ▶ To Database**.
2. The **Export to Database** dialog appears (Figure 9.4).

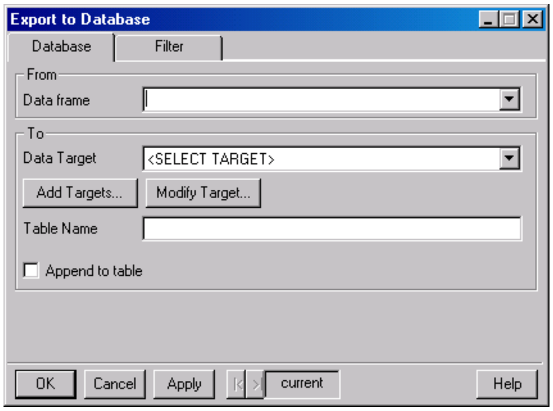


Figure 9.4: *The* ***Export to Database*** *dialog.*

3. Select a data frame object to export from the **Data frame** list.

4. Select a direct data source from the **Data Target** list.

   You can customize these or create new direct sources by clicking the **Add Targets** button. See below for further information on creating direct data sources.

   Choose one of these or select another one you have created.

   If you have not completely configured the source, the **Modify Data Source** dialog appears. Fill in all the fields with valid information for the data source chosen to continue.

5. Specify the table name you want to export to. Follow the syntax rules for table names that the target database imposes. Check your database documentation for more information on this topic. By default, the table name shown is based on the name of the data frame name specified in the dialog.

6. You can append data to the table you specify if it already exists in the database. To do this, check the **Append to table** checkbox.

Note

If you try to append data that does not have columns which match the data types of columns that already exist in the table, you will receive error messages and the export fails.

7. Click the **OK** button to perform the export.

You can add new direct data sources that are based on one of the four supplied direct database types, and you can add as many data sources as you wish. To add a direct data source:

1. From the **File** menu, select the **Export Data ▶ To Database**.

2. Click the **Add Targets** button and from the context menu which appears, choose **Add Direct Source**, and a new dialog appears.

3. Enter the name for the new direct data source in the **Name** field. The name you enter is used to display this data source in the list of data sources in the **Import From Database** and **Export to Database** dialogs. Choose a name that is different from other entries in the data sources list.

4. Select the type of database from the drop list of database choices in the field called **Type**.
5. Depending on the type you select, specify the username, password, server and database name using the fields provided. Some database types do not require server or database names and so that fields, so those fields may be unavailable for those types.
6. Click **OK** to add the source to the data sources list. When you add a source it is also selected as the current source to export to.

You can modify each of these with the appropriate information for your database configurations. To modify a direct data source:

1. From the **File** menu, select the **Export Data ▶ To Database**.
2. Select the direct data source from the **Data Target** list you wish to modify.
3. If the data source has invalid or incomplete information or you are using the data source for the first time, the **Modify Data Source** dialog appears.
4. If the **Modify Data Source** dialog does not appear, it means that the information for this data source is valid. Click the **Modify Target** button below the list to display the **Modify Data Source** dialog.
5. In the **Modify Data Source** dialog, specify your user name and password along with the server name and database name for the source as appropriate for the database type. The fields not required for a given type are unavailable ("grayed out") in the dialog.
6. You can also change the data source name shown in the **Name** field and the database type shown in the **Type** drop-down list. Changing the data source name changes how it is listed in the lists in the dialog for both the **Import From Database** and **Export to Database**. Changing which fields are available may require you to specify different information depending on the database type chosen. If you change the database type, it is a good idea to change the name to identify it as a different data source in the list.
7. Click the **OK** button to accept your changes.

You can also remove direct data sources. Use caution, as you can remove the four direct data sources that are provided with Spotfire S+. If you do, you can add them again following the procedures above to add a data source. To remove a data source:

1. From the **File** menu, select the **Export Data ▶ To Database**.
2. Select the direct data source from the **Data Target** list you wish to modify.
3. Click the **Modify Target** button, and the **Modify Direct Source** dialog appears.
4. Click the **Remove** button in this dialog to remove the data source.

How Direct Data Sources are Stored

The entries in the data sources drop lists appearing in the **Import From Database** and **Export to Database** dialogs are actually stored in a special text file located in the **.Prefs** subfolder of your project folder **[S_PROJ]\.Prefs**. The file is called **datasources.ini**, and can be edited with any text editor.

Each line of the file is a comma-delimited specification of the necessary information for the data source, as in the following example:

```
DIRECTDB:Direct DB2,direct-db2,,testdb,testqa,testqa
DIRECTDB:Direct Oracle,direct-oracle,ORACLE.TESTDB,,testqa,
DIRECTDB:Direct SQL Server,direct-sql,,,,
DIRECTDB:Direct Sybase,direct-sybase,qa.insightgul.com,,,
```

Each line must begin with the string `DIRECTDB`: This allows Spotfire S+ to distinguish the information as pertaining to direct data sources. Following this string, each field which appears in the dialog is entered in the following order, with commas separating the fields:

```
[name], [type], [server name], [database name], [username],
[password]
```

where `[name]` is the data source name that appears in the drop-down lists in the dialogs. The `[type]` field must be one of the following (matching the type field in the `importData` and `exportData` commands):

- `direct-db2`
- `direct-oracle`

- `direct-sql`
- `direct-sybase`

The `[password]` field must be specified as clear text.

Unspecified or blank field values must be separated by commas, as in the example below where the database name and password fields are left unspecified:

```
DIRECTDB:Direct Oracle,direct-oracle,ORACLE.TESTDB,,testqa,
```

As an alternative to managing direct database sources in the dialogs, you can simply edit this file after closing Spotfire S+. Restart Spotfire S+ so that the changes you made to the data sources in this file are used in Spotfire S+.

INSTALLING AND CONFIGURING DATABASE CLIENTS ON UNIX

In testing direct driver support on Linux® and UNIX® platforms, we have found that the database vendors provide fairly complete installation instructions regarding database clients on supported platforms.

Please refer to the installation instructions that came with the database software to install database clients on your system.

INSTALLING AND CONFIGURING DATABASE CLIENTS ON WINDOWS

In order to use direct database driver support in Spotfire S+ for Windows, you must install database client software on the same system where Spotfire S+ is installed. Currently, Spotfire S+ supports 32-bit versions of the following database clients:

- SQL Server 2000
- Sybase 12.5
- Oracle 9i
- DB2 7.2

The SQL Server 2000 client supports access to a variety of SQL Server versions, including SQL Server 6.5. The Oracle 9i client supports access to most previous versions of Oracle, including 8i.

Other versions of the database clients may work but have not been tested by TIBCO Software Inc..

You can follow the instructions provided by a database vendor to install a database client. Alternatively, you can follow the steps below for the appropriate client. These instructions are provided to help get you started quickly installing and using a particular database client. For more in-depth information, consult the instructions provided with the database client software.

SQL Server 2000 Client

1. Insert the Microsoft SQL Server 2000 Enterprise Edition CD in your CD-ROM drive and run **English ▶ Ent ▶ autorun.exe**.
2. In the dialog that appears, select **SQL Server 2000 Components** from the available setup options.
3. In the next screen, select to install the **Database Server**.
4. In the next screen, select **local computer**, and select a destination on your system.
5. In the next screen, select **Create a new installation of SQL Server, or Install Client Tools** from the available options.
6. In the next screen, choose **Client Tools Only** from the available options
7. In the feature installation screen, accept all the selected features – *do not remove any*!
8. Once installed, run the **Enterprise Manager** from the **Start** menu using the icon at **Programs ▶ Microsoft SQL Server ▶ Enterprise Manager**.
9. In the **SQL Server Enterprise Manager** window, there is a sub-window called **Console Root ▶ Microsoft SQL Servers**. In the left pane of this window, expand the tree view to **Console Root ▶ Microsoft SQL Servers ▶ SQL Server Group**.
10. Right-click this expanded node and select **New SQL Server Registration** from the menu.

11. In the **Select a SQL Server** dialog which appears, choose the name of the server system you installed SQL Server on from the list on the left, and click the **Add** button to add it to the right.
12. In the dialog **Select an Authentication Mode**, choose the radio button for **SQL Server Authentication**.
13. Next, specify the appropriate login name and password when prompted.
14. Finally, choose to add the SQL server to the existing group called **SQL Server Group**.

After you have successfully setup the client following the steps above, ensure that the paths below are in your PATH environment variable:

```
[install path]\80\Tools\BINN
```

where `[install path]` is the path you chose to install the client tools in step 4 above.

Sybase Client

Insert the Sybase 12.5 Adaptive Enterprise client CD into your CD-ROM drive. The installation program should automatically start.

Note

If the Sybase installer crashes or locks up during startup in **java.exe**, then you may need to disable Java "just-in-time" compiling on your system. See Sybase technical article **www.sybase.com/detail/1,6904,1013241,00.html** on the Sybase Web site for more information about this problem.

Carry out the following steps to install a Sybase client on your system:

1. Choose the **Standard** install.
2. Accept the default location or specify another location. Make sure that the install directory you choose does *not* contain any spaces.
3. Ensure that the following paths are in your `PATH` environment value:

```
[install path]\CFG-1_0\bin;[install path]\OCS-
12_5\dll;[install path]\OCS-12_5\lib3p;[install
path]\OCS-12_5\bin
```

where `[install path]` is the path you chose in step 2 above.

After installation, select **Programs ▶ Sybase ▶ dsedit** from the **Start** menu to start this utility program. Then do the following steps:

1. Click **OK** on the first screen to open the **Interfaces Driver** screen.
2. From the **Server Object** menu, select **Add**.
3. In the **Input Server Name** box, enter the network internet protocol name of the server running Sybase, and click **OK** (e.g., **qaimage.tibco.com**).
4. In the **attributes** column of the **Interfaces Driver** dialog, double-click the server address row.
5. In the **Network Address Attribute** window, click the **Add** button.
6. Select **TCP** as the network connection protocol from the drop-down list and enter

   [server ip name], 2048

in the **Edit** field. **[server ip name]** is the internet protocol name of the server that has Sybase installed, and should be the same as specified in step 3 above, as in **qaimage.tibco.com**. 2048 is the port number it receives on, and you may have to change the port number, depending on the server. Click **OK** to accept the changes.

The **[server ip name]** you specify here is used in Spotfire S+ to connect to the server and use Sybase.

7. Click **OK** to accept the **Network Address Attribute** window.
8. Test the connection to the new server by selecting **Ping** from the **Server Object** menu.
9. In the **Ping** dialog, click the **Ping** button. You should see another dialog appear, indicating that the connection was successful.
10. Close the **dsedit** utility program.
11. Test the connection using the Sybase ISQL utility:
    - Open a DOS Command window and type the following:

isql –Utestqa –Ptestqa – S[server ip name]

- At the isql program prompt, enter the following:

  select * from pubs2.dbo.sales
  go

After you type **go**, you should get an output table printed in the window.

Oracle Client

Insert the Oracle 9i Client for Win32 CD in your CD-ROM drive. The setup starts automatically, and you can follow these steps:

1. In the **Oracle Universal Installer: File Locations** dialog, enter the path to install the Oracle client software on your system in the **Destination path** field.
2. For the type of installation, select **Administrator**.
3. Use the default port number of 2030.
4. After installation, the **Oracle Net Configuration Assistant** automatically appears. If it does not automatically appear, you can manually start it from the **Start** menu at **Programs ▶ Oracle - OraHome90 ▶ Configuration and Migration Tools ▶ Net Configuration Assistant**.
5. In the **Net Configuration Assistant Wizard**, enter the values specified in the appropriate wizard steps indicated below:
   - Select **Perform typical configuration** in the **Welcome** step. Click **Next**.
   - Select **No, I will create net service names myself...**. Click **Next**.
   - Select **Oracle8i or later database or service**. Click **Next**.
   - Specify an appropriate network service name. It is suggested that you specify a name that contains the name of the database and your network domain, as in **testdb.tibco.com** for the service name. Click **Next**.
   - Select **TCP** as the protocol. Click **Next**.

- Specify the name of the server which is running Oracle on your network. Specify only the system name (and not an IP address), such as **qadb-s2k** for the host name, and specify an appropriate port number. You can accept **1521** as the default port or change it, depending on how your server is configured. Click **Next**.
- Select **Yes, perform a test** to test the connection. Click **Next**. You should get an unsuccessful connection screen with a **Change Login** button on it.
- Click the **Change Login** button, and specify the appropriate username and password in the dialog. Click **OK** and then **Next**.

6. Now, the test connection should report success. Click **Next**.
7. Accept or change the network service name shown. This is the name you specify in Spotfire S+ to access the ORACLE server. Click **Next**.
8. When asked whether you want to configure another net service name, choose **No**. Click **Next**.

After a successful installation, ensure that the following paths are in your PATH environment value:

```
[install path]\ora90\bin
```

where `[install path]` is the path you chose during setup in step 1 above.

DB2 Client

Insert the DB2 Universal Database Enterprise Edition Version 7.2 CD into your CD-ROM drive. The installer should start automatically.

1. Choose **Install** from the list of setup options.
2. In the **Select Products** dialog, choose **DB2 Administration Client**.
3. Select **Typical** setup type.
4. Select a destination for the installation or accept the default path.

5. In the **Enter Username and Password for Control Center Server** dialog, enter your network login name and password. Note that you can enter another username and password, but using your network login and password make it easy to remember.
6. You may receive a dialog telling you that you don't have privileges to do certain things with DB2 on your system. Click the **OK** button in this dialog to continue with the setup.
7. If you are prompted to restart your system, make sure you do this. Some services that are installed as part of the DB2 installation need to be installed and mounted.
8. After your system is restarted or after the end of a successful setup, the **First Steps** dialog appears. In this dialog, select **Catalog Sample Databases** from the list on the left.
9. The **Client Configuration Assistant** is started.
10. In the **Welcome** dialog, click the **Add Database** button.
11. A "wizard" dialog appears. In the **Source** page, click the **Search the network** radio button, and switch to the **Database name** page.
12. In the **Tree** view, expand the **Other Systems** node and wait until the program has scanned all network systems for DB2 servers. The list it finds appears below this node.
13. Look for the name of the server that has DB2 installed on it in this tree. If you don't see it, check the server, and make sure DB2 has been properly started. Once it is located, expand the server name node in the tree to see a list of databases.
14. Select the database you want to use from the **Local databases** node. The name appears in the **Target Database** field at the bottom of the dialog. Click **Next**.
15. In the **Alias** page, verify that the alias for this database is listed as the database name you chose. Click **Next**.
16. In the **ODBC** page, accept the defaults for registering the database as a system source. Click **Finish**.

Click the **Close** button when a confirmation dialog appears. Do not attempt to test the connection at this time as the settings are not correct yet. You are then returned to the **Client Configuration Assistant** main window.

17. You can see that the database you selected was added as a database. Select this database, then click the **Properties** button. This opens another dialog called **Database Properties – [database name]**.
18. Click the **Properties** button. This opens another dialog called **Update Connection Wizard – [database name]**.
19. Protocol should be set to **TCP/IP**. Click **Next**.
20. Correct the host name by changing it to the IP name of the database server, such as **qadb-snt.tibco.com**. Click the **Finish** button. You are returned to the **Database Properties – [database name]** dialog. Click **OK**.
21. In the main **Client Configuration Assistant** window, select the database you selected from the list and click the **Test** button.
22. In the **Connect to DB2 Database**, enter the appropriate username and password in the fields and leave other default settings. Click the **OK** button. You should receive a "connection successful" dialog.
23. Close the **Client Configuration Assistant**.
24. Close the **First Steps** dialog.
25. Ensure that the path `[install path]\BIN` is in your `PATH` environment value. `[install path]` is the path you selected in step 4 above.

EXPORTING DATA

Using the `exportData` Function

You use the `exportData` function to export Spotfire S+ data objects to formats for applications other than Spotfire S+. (To export data for use by Spotfire S+, use the `data.dump` function–see page 325.) You can invoke `exportData` from either the Spotfire S+ prompt or the **File ▶ Export Data** menu option.

When exporting to most file types with `exportData`, 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 is exported into a new data file using default settings. For greater control, you can specify your own settings by using additional arguments to `exportData`. Table 9.8 lists the arguments to the `exportData` function.

Table 9.8: *Arguments to* `exportData`.

Argument	Required or Optional	Description
`data`	Required	The data frame or matrix to be exported.
`file`	Required	A character string specifying the name of the export file to create.
`type`	Optional	A character string specifying the file type of the export file. See the "Type" column of Table 9.1 for a list of possible values.
`keep`	Optional	A character vector of variable names, or a numeric vector of column numbers, specifying which variables are to be exported. Only one of `keep` or `drop` may be specified.
`drop`	Optional	A character vector of variable names, or a numeric vector of column numbers, specifying which variables are *not* to be exported. Only one of `keep` or `drop` may be specified.

Table 9.8: *Arguments to* `exportData`. *(Continued)*

Argument	Required or Optional	Description
`filter`	Optional	A character string containing a logical expression for selecting the rows to be exported. For details, see Filter Expressions on page 289.
`format`	Optional	A single character string specifying the format for each field when exporting to a formatted ASCII (FASCII) text file. For details, see Notes on Importing Files of Certain Types on page 291.
`delimiter`	Optional	A character string specifying the delimiter to use. The default is a blank space (`" "`). This argument is used only when exporting to ASCII text files.
`colNames`	Optional	A logical flag. If `TRUE`, column names are also exported.
`rowNames`	Optional	A logical flag. If `TRUE`, row names are also exported.
`quote`	Optional	A logical flag. If `TRUE`, quotes are placed around character strings. The default is `TRUE`.
`odbcConnection`	Required if `type="ODBC"`	An encrypted character string containing the ODBC connection string.
`odbcTable`	Required if `type="ODBC"`	The name of the ODBC table to be created.
`time.out.format`	Optional	A character string specifying the format to use when exporting date/time data to ASCII or FASCII text files.

Other Data Export Functions

In addition to the exportData function, Spotfire S+ provides several other functions for exporting data, discussed below.

The data.dump Function

When you want to share your data with another Spotfire S+ user, you can export your data to a Spotfire S+ file format by using the data.dump function:

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

By default, the data object matz is exported to the file **dumpdata** in your Spotfire S+ start-up folder (Windows) or directory (UNIX). You can specify a different output file with the connection argument to data.dump:

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

Hint

The connection argument needn't specify a file; it can specify any valid Spotfire S+ connection object.

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

```
> data.dump("halibut", where="data")
```

The cat and write Functions

The inverse operation to the scan function is provided by the cat and write functions. The result of either cat or write is just an ASCII file with data in it; there is no Spotfire S+ structure written to the file. 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.

The cat function is a general-purpose writing tool in Spotfire S+, 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)
```

Note

The files written by cat and write do not contain Spotfire S+ structure information. To read them back into Spotfire S+, you must reconstruct this information.

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 Spotfire S+, first transform the matrix 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 UNIX pager; it contains the following three lines:

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

The write.table Function

The inverse operation to read.table is provided by write.table. The write.table function can be used to export a data frame into an ASCII text file:

```
> write.table(fuel.frame, "fuel.txt")
```

EXPORTING GRAPHS

The `export.graph` function is only available on Windows, and is used to export a graph named `Name` to the file `FileName` using the file format specified by `ExportType`. Table 9.9 lists the arguments to the `export.graph` function.

Table 9.9: *Arguments to* `export.graph`.

Argument	Required or Optional	Description
`FileName`	Required	A character string specifying the name of the file to be created. If a file by this name already exists, it is overwritten.
`Name`	Optional	A character string specifying the object path name for a graphsheet. The default uses the graphsheet that is currently active. If no graphsheet is active, then `Name` is required.
`ExportType`	Optional	A character string specifying the file type of the exported graph. For a complete discussion of this argument, see page 328.
`Qfactor`	Optional	An integer value that determines the degree of loss in the compression process. For a complete discussion of this argument, see page 330.
`ColorBits`	Optional	An integer value that specifies the color bits value used when saving an image. For a complete discussion of this argument, see page 330.
`Height`	Optional	A numeric value that specifies the height of the output image. This argument accepts any floating point value. The default of `-1` causes the graph page height to be used.

Table 9.9: *Arguments to* `export.graph`*. (Continued)*

Argument	Required or Optional	Description
`Width`	Optional	A numeric value that specifies the width of the output image. This argument accepts any floating point value. The default of `-1` causes the graph page width to be used.
`Units`	Optional	A character string that specifies the units of the `Height` and `Width` arguments. Recognized values are `"inch"` and `"cm"`; any other input is interpreted as `"inch"`, the default value.

Specifying the `ExportType` Argument

Some of the most common values for the `ExportType` argument include `"BMP"`, `"WMF"`, `"EPS"`, `"EPS TIFF"`, `"TIF"`, `"GIF"`, `"JPG"`, `"PNG"`, `"IMG"`, `"EXIF"`, `"PCT"`, `"TGA"`, and `"WPG"`. If this argument is not specified, the file type is inferred from the extension used in the `FileName` argument.

Note

If you are running the Spotfire S+ Windows graphical user interface (which is 32-bit only) on a 64-bit version of Windows, you cannot export GIF images. If you call the S-PLUS function `export.graph` and specify a GIF file or export type, it returns an error; if you use the Spotfire S+ GUI export dialog box, notice that the GIF option is not available in the drop-down list.

Table 9.10 describes the map between file extensions and file types. If `FileName` does not include an extension from Table 9.10, one is added based on the value of this argument. To export a graph to a file that does not have an extension, specify the appropriate `ExportType` format and end the `FileName` character string with a period.

Table 9.10: *Map between file extensions and file types for the* `ExportType` *argument.*

Extension	`ExportType` **Setting**	**File Format**
.bmp	`BMP`	Windows Bitmap, with no compression
.cal	`CAL`	CALS Raster file
.cmp	`CMP`	LEAD Compression Format
.emf	`EMF`	Windows Enhanced MetaFile
.eps	`EPS`	Encapsulated PostScript
.fax	`FAX`	Raw FAX, compressed using CCITT group 3, 1 dimension
.gif	`GIF`	CompuServe GIF (requires that LZW compression be enabled)
.ica	`ICA`	IOCA, compressed using CCITT group 3, 1 dimension
.img	`IMG`	GEM Image
.jpg	`JPG`	JPEG File Interchange Format with YUV 4:4:4 color space
.mac	`MAC`	MacPaint
.msp	`MSP`	Microsoft Paint
.pct	`PCT`	MacPic

Table 9.10: *Map between file extensions and file types for the* `ExportType` *argument. (Continued)*

Extension	`ExportType` Setting	File Format
.pcx	`PCX`	ZSoft PCX
.png	`PNG`	Portable Network Graphics
.psd	`PSD`	Adobe Photoshop 3.0
.ras	`RAS`	Sun Raster file
.tga	`TGA`	TrueVision TARGA
.tif	`TIF`	Tagged Image File Format, with no compression and with RGB color space
.wfx	`WFX`	Winfax, compressed using CCITT group 3, 1 dimension
.wmf	`WMF`	Windows MetaFile
.wpg	`WPG`	Word Perfect

Specifying the `QFactor` Argument

The `QFactor` argument is a number that determines the degree of loss in the compression process when saving an image file to the following `ExportType` formats: `"CMP"`, `"JPG"`, `"JPG YUV4"`, `"JPG YUV2"`, `"JPG YUV1"`, `"TIF JPG"`, `"TIF JPG YUV4"`, `"TIF JPG YUV2"`, `"TIF JPG YUV1"`, and `"EXIF JPG"`. The valid range is from `2` to `255`, with `2` resulting in perfect quality and `255` resulting in maximum compression. The default value is `2`.

Note

The effect of this argument is identical to the "quality" parameter (0-100%) used in most applications that view and convert JPEG graphics.

Specifying the `ColorBits` Argument

Valid options for each format are listed in Table 9.11. The default is to use the maximum value supported by the requested format. This argument is ignored for the following `ExportType` formats: `"EMF"`, `"EPS"`, `"EPS TIFF"`, `"EPS WMF"`, and `"WMF"`.

Table 9.11: *Valid options for the `ColorBits` argument.*

`ExportType` **Setting**	**Format Description**	`ColorBits` **Setting**
JPEG and LEAD Compressed		
"CMP"	LEAD Compression Format	8, 24
"JPG" or "JPG YUV4"	JPEG File Interchange Format with YUV 4:4:4 color space	8, 24
"JPG YUV2"	JPEG File Interchange Format with YUV 4:2:2 color space	8, 24
"JPG YUV1"	JPEG File Interchange Format with YUV 4:1:1 color space	8, 24
Compressed TIFF		
"TIF JPG" or "TIF JPG YUV4"	Tagged Image File with JPEG compression and YUV 4:4:4 color space	8, 24
"TIF JPG YUV2"	Tagged Image File with JPEG compression and YUV 4:2:2 color space	8, 24
"TIF JPG YUV1"	Tagged Image File with JPEG compression and YUV 4:1:1 color space	8, 24
"TIF PACK"	Tagged Image File with PackBits compression and RGB color space	1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32
"TIF PACK CMYK"	Tagged Image File with PackBits compression and CMYK color space	24, 32
"TIF PACK YCC"	Tagged Image File with PackBits compression and YCbCr color space	24
"CCITT"	TIFF, compressed using CCITT	
"CCITT G3 1D"	TIFF, compressed using CCITT, group 3, 1 dimension	
"CCITT G3 2D"	TIFF, compressed using CCITT, group 3, 2 dimensions	
"CCITT G4"	TIFF, compressed using CCITT, group 4	
TIFF Without Compression		
"TIF"	Tagged Image File Format, with no compression and with RGB color space	1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32
"TIF CMYK"	Tagged Image File Format, with no compression and with CMYK color space	24, 32
"TIF YCC"	Tagged Image File Format, with no compression and with YCbCr color space	24

Table 9.11: *Valid options for the `ColorBits` argument. (Continued)*

`ExportType` **Setting**	**Format Description**	`ColorBits` **Setting**
BMP Formats		
"BMP	Windows BMP, with no compression	1, 4, 8, 16, 24, 32
"BMP RLE"	Windows BMP, with RLE compression	4, 8
"OS2"	OS/2 BMP version 1.x	1, 4, 8, 24
"OS2 2"	OS/2 BMP version 2.x	1, 4, 8, 24
Exif Formats		
"EXIF"	Exif file containing a TIFF image, no compression with RGB color space	24
"EXIF YCC"	Exif file containing a TIFF image, no compression with YCbCr color space	24
"EXIF JPG"	Exif file containing a JPEG compressed image	24
"EXIF 411"	Exif 2.0 file containing a JPEG compressed image	24
Other Color Formats		
"PCX"	ZSoft PCX	1, 4, 8, 24
"WMF"	Windows MetaFile	24
"EMF"	Windows Enhanced MetaFile	24
"PSD"	Adobe Photoshop 3.0	1, 8, 24
"PNG"	Portable Network Graphics	1, 4, 8, 24
"TGA"	TrueVision TARGA	8, 16, 24, 32
"EPS"	Encapsulated PostScript	24
"EPS TIFF"	Encapsulated PostScript with TIFF header	24
"EPS WMF"	Encapsulated PostScript with WMF header	24
"RAS"	Sun Raster	1, 4, 8, 24, 32
"WPG"	Word Perfect (raster only)	1, 4, 8
"PCT"	MacPict	1, 4, 8, 24
Formats requiring LZW compression to be enabled		
"TIF LZW"	Tagged Image File Format with LZW compression and RGB color space	1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32
"TIF LZW CMYK"	Tagged Image File Format with LZW compression and RGB color space	24, 32
"TIF LZW YCC"	Tagged Image File Format with LZW compression and RGB color space	24
"GIF"	CompuServe GIF	1, 2, 3, 4, 5, 6, 7, 8

Table 9.11: *Valid options for the `ColorBits` argument. (Continued)*

`ExportType` **Setting**	**Format Description**	`ColorBits` **Setting**
<u>1-Bit FAX Formats</u>		
"FAX" or "FAX G3 1D"	Raw FAX, compressed using CCITT group 3, 1 dimension	1
"FAX G3 2D"	Raw FAX, compressed using CCITT group 3, 2 dimensions	
"FAX G4"	Raw FAX, compressed using CCITT group 4	
"WFX" or "WFX G3"	Winfax, compressed using CCITT group 3, 1 dimension	
"WFX G4"	Winfax, compressed using CCITT group 4	
"ICA" or "ICA G3 1D"	IOCA, compressed using CCITT group 3, 1 dimension	
"ICA G3 2D"	IOCA, compressed using CCITT group 3, 2 dimensions	
"ICA G4"	IOCA, compressed using CCITT group 4	
"ICA RAW" or "ICA RAW G3 1D"	IOCA, compressed using CCITT group 3, 1 dimension, without the MO:DCA wrapper	
"ICA RAW G3 2D"	IOCA, compressed using CCITT group 3, 2 dimensions, without the MO:DCA wrapper	
"ICA RAW G4"	IOCA, compressed using CCITT group 4, without the MO:DCA wrapper	
"CAL"	CALS Raster file	
<u>Other 1-Bit Formats</u>		
"MAC"	MacPaint	1
"MSP"	Microsoft Paint	
"IMG"	GEM Image	

CREATING HTML OUTPUT

Spotfire S+ provides a variety of tools for generating HTML output. In this section, we discuss 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 contains 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 Spotfire S+ 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 Spotfire S+ 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.

USING DIRECT DATABASE DRIVERS 10

OVERVIEW

Note

As of Spotfire S+ version 8.2, native database drivers are deprecated. In lieu of these drivers, you should use JDBC/ODBC drivers for all supported database vendors.

Spotfire S+ supports access to the following databases via direct database drivers:

- Microsoft SQL Server® (Windows® only).
- IBM DB2® (Windows, Solaris® 32, Linux®, Compaq Tru64®, HP®, AIX®).
- Sybase® (Windows, Solaris 32, HP, AIX).
- Oracle® (Windows, Solaris 32, Linux, HP, AIX).

We strongly recommend using ODBC and JDBC drivers.

You can use the `importData` and `exportData` commands to access the above databases using the new direct drivers.

Exporting to databases now allows existing tables to be either replaced or appended to.

In addition, in Spotfire S+ for Windows, you can use the **Import From Database** and **Export to Database** dialogs to access these databases as data sources, just like accessing ODBC data sources. The previous versions of Spotfire S+ had menu items such as **Import Data ▶ From ODBC Connection** and **Export Data ▶ To ODBC Connection**, and these have been replaced with **Import From Database** and **Export to Database**, respectively. The dialogs presented now allow you to select either ODBC or direct database sources from the same list.

Direct database access is accomplished by using driver components which must be separately installed on the system that is running Spotfire S+. These driver components are provided by the database vendor and usually consist of components that can be called directly by Spotfire S+ to send and receive database information in the native format that the database supports. In many cases, direct database drivers provide faster connectivity and data transfer than ODBC because there are fewer layers of data translation and interpretation of the request than with ODBC.

SPOTFIRE S+ COMMANDS FOR IMPORTING AND EXPORTING

Before you can use the importData or exportData commands to access database via the direct drivers, you must install database client software on your system. Please refer to the sections below on installing database clients for your system type.

There are four new database type keywords that can be used in the type parameter of the **importData** or **exportData** commands:

- DIRECT-DB2
- DIRECT-ORACLE
- DIRECT-SQL
- DIRECT-SYBASE

As an example, consider the following Spotfire S+ commands to send data to a Sybase database:

```
mydata <- data.frame(COL1=c(1.2,1.3,1.51,2.1,3.9),
     COL2=c("a", "b", "c", "d", "e"),
     COL3=timeDate(c("1/1/2003", "2/1/2003",
            "3/15/2005", "10/24/2003", "11/11/2004"),
            format="%02m/%02d/%Y %02H:%02M:%02S.%03N"))

exportData(mydata, type="DIRECT-SYBASE",
     user="testqa", password="testqa",
     server="qaimage.tibco.com", database="testdb",
     table="testDirectSybase", appendToTable=F)
```

In this example, the data frame mydata is exported to the table testDirectSybase into the database testdb on the server qaimage.tibco.com. The database client software validates the user and password parameter values prior to exporting the data, and if they are incorrect, an error is reported in Spotfire S+. The server name you specify here should be the one you specified during installation of the Sybase client software. See Step 6 in the Windows Sybase client installation instructions below for further information on this.

Notice that the string `DIRECT-SYBASE` was used in the `type` parameter of the `exportData` command to specify connection to a Sybase database. Also note that the user name, password, server name, database and table name are specified. For each of the four databases supported by direct drivers in Spotfire S+, slightly different combinations of these parameters must be specified. See the table below for a list of the differences.

Table 10.1: *Table of parameters required for various direct database types in `importData` and `exportData`.*

Database type	**Required parameters for** `importData and exportData`	**Comments**
`DIRECT-DB2`	`user, password, database, table`	Server parameter should *not* be specified for DB2.
`DIRECT-ORACLE`	`user, password, server, table`	Database parameter should not be specified for Oracle. Server parameter should be the **network service name** you specified when installing the Oracle client software. See Step 5. in the section Oracle Client.
`DIRECT-SQL`	`user, password, server, database, table`	Server parameter should be the server name you specified in the **SQL Server Enterprise Manager** program. See Step 11. in the section SQL Server 2000 Client.
`DIRECT-SYBASE`	`user, password, server, database, table`	Server parameter should be the server name you specified in the Sybase installation. See Step 6. in the section Sybase Client.

Notice the existence of the `appendToTable` parameter, which controls whether or not to append the data you are exporting to the specified table. If this parameter is false, the data overwrites the table specified; if true, the data is appended.

An error occurs if the data types of the data you are sending and those already present in the table do not match. For example, if you export strings to a table that currently has columns of numeric values, you receive an error and the export fails.

Here is an example of using the `importData` command to read data into Spotfire S+ from an Oracle database table via the direct drivers:

```
mynewdata <- importData(type="direct-oracle",
               user="testqa", password="testqa",
               server="ORACLE.TESTDB",
               table="testDirectOracle")
```

In this example, the table `testDirectOracle` is used. Since no specific SQL query is specified (normally specified with the `sqlQuery` parameter), all data from the table is imported. The database client software validates the user and password parameter values prior to importing the data and if they are incorrect, an error is reported in Spotfire S+. The server name you specify here should be the one you specified as the network service name during installation of the Oracle client software. See Step 5. in the section Oracle Client installation instructions below for further information on this.

For more information on using the `importData` and `exportData` commands as well as additional information on using the `sqlQuery` parameter in `importData`, please see the online help.

DIALOGS FOR IMPORTING AND EXPORTING

In Spotfire S+ for Windows, you can use the **Import From Database** and **Export to Database** dialogs to access databases as data sources, just like accessing ODBC data sources. The previous versions of Spotfire S+ had menu items such as **Import Data ▶ From ODBC Connection** and **Export Data ▶ To ODBC Connection**, and these have been replaced with **Import From Database** and **Export to Database**, respectively. The dialogs presented now allow you to select either ODBC or direct database sources from the same list.

Import From Database

You can use the Spotfire S+ **Import From Database** dialog to import table data from direct database sources available on your system. Only those sources for which you have installed database clients are supported. For more information on installing database client software, see the section Install and Configure Database Clients (UNIX) and the section Install and ConfigurE Database Clients (Windows) later in this document.

Spotfire S+ lists all data sources it supports in the **Import From Database** dialog. However, only those sources which have database clients installed on your system work. The others report errors until you install and configure the appropriate database clients.

When you first run Spotfire S+, four direct data sources are listed in the **Import From Database** dialog, one for each type supported on Windows:

- Direct DB2
- Direct Oracle
- Direct SQL Server
- Direct Sybase

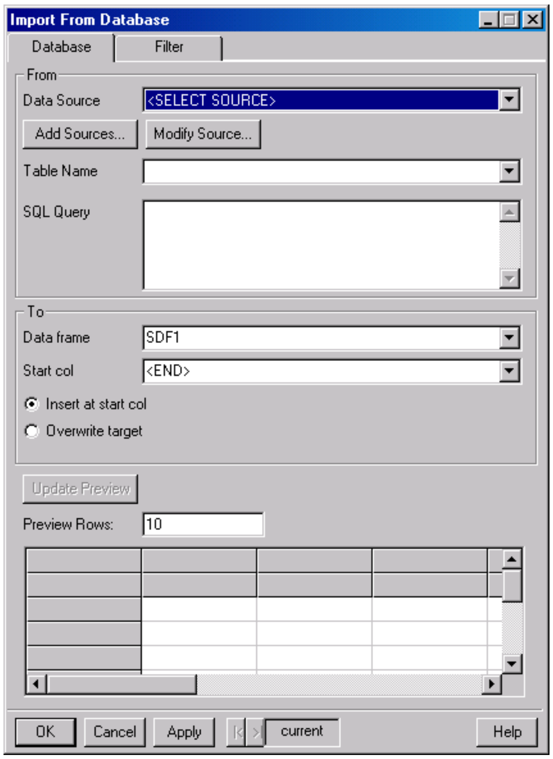
You can modify these data sources to configure them for your particular database setup, such as setting the correct server, database, username, password, and other information appropriate for how the database is set up on your network.

You can also add new direct sources (based on one of the four types listed above) to the list of data sources. This allows you to have one or more data sources for the same database type, thus enabling you to specify different database names or usernames and passwords to access other sets of tables on the same database server.

You can also remove direct data sources from the **Data Source** list.

To import data from a direct database source:

1. From the **File** menu, select **Import Data ▶ From Database**.
2. The **Import From Database** dialog appears, as shown in Figure 10.1.



Import From Database

Database | Filter

From

Data Source: <SELECT SOURCE>

Add Sources... | Modify Source...

Table Name:

SQL Query:

To

Data frame: SDF1

Start col: <END>

(●) Insert at start col

( ) Overwrite target

Update Preview

Preview Rows: 10

OK | Cancel | Apply | current | Help

Figure 10.1: *The **Import From Database** dialog.*

3. Select a direct data source from the **Data Source** list.

   You can customize these or create new direct sources by clicking the **Add Sources** button. See below for further information on creating direct data sources.

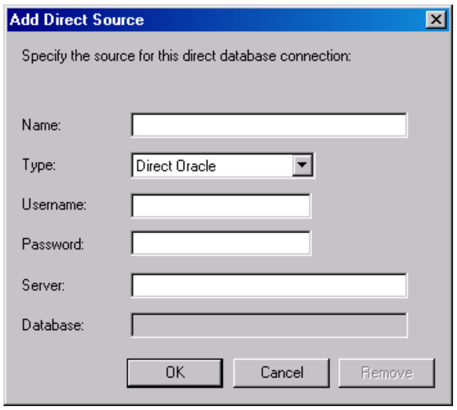
Choose one of these or select another one you have created.

If you have not completely configured the source, the **Modify Data Source** dialog appears. Fill in all the fields with valid information for the data source chosen to continue.

4. Once a direct data source has been selected, the **Tables** list changes to contain all the tables in that source. Select a table from the list.
5. Specify any other options, including a valid SQL query in the **SQL Query** field. If you leave the **SQL Query** field blank, a default query of all columns and rows from the selected table is performed.
6. Click the **OK** button to start the import.

You can add new direct data sources that are based on one of the four supplied direct database types. You can add as many data sources as you wish. To add a direct data source, do the following:

1. From the **File** menu, select **Import Data ▶ From Database**.
2. Click the **Add Sources** button, and from the context menu, select **Add Direct Source**.
3. The **Add Direct Source** dialog appears (Figure 10.2).


Add Direct Source

Specify the source for this direct database connection:

Name:
Type: Direct Oracle
Username:
Password:
Server:
Database:

Figure 10.2: *The **Add Direct Source** dialog.*

4. Enter the name for the new direct data source in the **Name** field. The name you enter is used to display this data source in the list of data sources in the **Import From Database** and **Export To Database** dialogs. Choose a name that is different from other entries in the data sources list.

5. Select the type of database from the drop list of database choices in the **Type** field.

6. Depending on the type you select, specify the username, password, server and database name using the fields provided. Some database types do not require a server or database name, and so those fields may be unavailable.

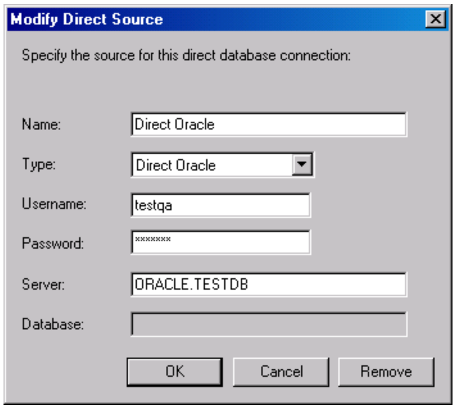
7. Click **OK** to add the source to the data sources list. When you add a source, it is also selected as the current source to import from.

You can modify each of these with the appropriate information for your database configurations. To modify a direct data source:

1. From the **File** menu, select **Import Data ▶ From Database**.

2. Select the direct data source from the **Data Sources** list you want to modify.

3. If the data source has invalid or incomplete information or you are using the data source for the first time, the **Modify Data Source** dialog appears, as shown in Figure 10.3.


Modify Direct Source

Specify the source for this direct database connection:

Name: Direct Oracle
Type: Direct Oracle
Username: testqa
Password: *******
Server: ORACLE.TESTDB
Database:

Figure 10.3: *The **Modify Direct Source** dialog.*

4. If the **Modify Data Source** dialog does not appear, it indicates that the information for this data source is valid. Click the **Modify Source** button below the list to display the **Modify Data Source** dialog.
5. In the **Modify Data Source** dialog, specify your user name and password along with the server name and database name for the source as appropriate for the database type. See Table 10.1 in the section Spotfire S+ Commands for Importing and Exporting for help identifying which fields need to be filled out for a given database type. The fields not required for a given type are greyed out in the dialog.
6. You can also change the data source name shown in the **Name** field and the database type shown in the **Type** drop list. Changing the data source name changes how it is listed in the **Data Sources** list in the dialog for both the **Import** and **Export** dialogs. Changing the database type changes which fields are available and may require you to specify different information depending on the database type chosen. If you change the database type, it is a good idea to change the name to identify it as a different data source in the list.
7. Click the **OK** button to accept your changes.

You can also remove direct data sources. Be careful using this dialog, as you can remove the four direct data sources that are provided with Spotfire S+. If you do, you can add them again following the procedures above to add a data source. To remove a data source:

1. From the **File** menu, select **Import Data ▶ From Database**.
2. Select the direct data source from the **Data Sources** list you want to modify.
3. Click the **Modify Source** button. The **Modify Direct Source** dialog appears.
4. Click the **Remove** button in this dialog to remove the data source.

Export to Database

You can use the Spotfire S+ **Export to Database** dialog to export data frame objects from Spotfire S+ to direct database sources available on your system. Only those sources that you have installed database clients for are supported. For more information on installing database client software, see the Installing and Configuring sections later in this document.

Spotfire S+ lists all data sources it supports in the **Export to Database** dialog. However, only those sources which have database clients installed on your system work. The others sources report errors until you install and configure the appropriate database clients.

When you first run Spotfire S+, four direct data sources are listed in the **Export To Database** dialog, one for each type supported on Windows. These are listed as follows:

- Direct DB2
- Direct Oracle
- Direct SQL Server
- Direct Sybase

You can modify these data sources to configure them for your particular database setup, such as setting the correct server, database, username, password, and other information appropriate for how the database is setup on your network.

You can also add new direct sources (based on one of the four types listed above) to the list of data sources. This allows you to have one or more data sources for the same database type, thus enabling you to specify different database names or usernames and passwords to access other sets of tables on the same database server. You can also remove direct data sources from the data sources list.

To export data to a direct database source:

1. From the **File** menu, select the **Export Data ▶ To Database**.

2. The **Export to Database** dialog appears (Figure 10.4).

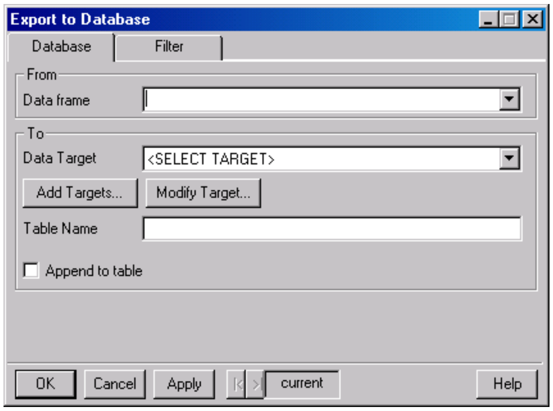


Figure 10.4: *The **Export to Database** dialog.*

3. Select a data frame object to export from the **Data frame** list.
4. Select a direct data source from the **Data Target** list.

   You can customize these or create new direct sources by clicking the **Add Targets** button. See below for further information on creating direct data sources.

   Choose one of these or select another one you have created.

   If you have not completely configured the source, the **Modify Data Source** dialog appears. Fill in all the fields with valid information for the data source chosen to continue.
5. Specify the table name you want to export to. Follow the syntax rules for table names that the target database imposes. Check your database documentation for more information on this topic. By default, the table name shown is based on the name of the data frame name specified in the dialog.

6. You can append data to the table you specify if it already exists in the database. To do this, check the **Append to table** checkbox.

Note

If you try to append data that does not have columns which match the data types of columns that already exist in the table, you will receive error messages and the export fails.

7. Click the **OK** button to perform the export.

You can add new direct data sources that are based on one of the four supplied direct database types, and you can add as many data sources as you wish. To add a direct data source:

1. From the **File** menu, select the **Export Data ▶ To Database**.
2. Click the **Add Targets** button and from the context menu which appears, choose **Add Direct Source**, and a new dialog appears.
3. Enter the name for the new direct data source in the **Name** field. The name you enter is used to display this data source in the list of data sources in the **Import From Database** and **Export to Database** dialogs. Choose a name that is different from other entries in the data sources list.
4. Select the type of database from the drop list of database choices in the field called **Type**.
5. Depending on the type you select, specify the username, password, server and database name using the fields provided. Some database types do not require server or database names and so that fields, so those fields may be unavailable for those types.
6. Click **OK** to add the source to the data sources list. When you add a source it is also selected as the current source to export to.

You can modify each of these with the appropriate information for your database configurations. To modify a direct data source:

1. From the **File** menu, select the **Export Data ▶ To Database**.

2. Select the direct data source from the **Data Target** list you wish to modify.
3. If the data source has invalid or incomplete information or you are using the data source for the first time, the **Modify Data Source** dialog appears.
4. If the **Modify Data Source** dialog does not appear, it means that the information for this data source is valid. Click the **Modify Target** button below the list to display the **Modify Data Source** dialog.
5. In the **Modify Data Source** dialog, specify your user name and password along with the server name and database name for the source as appropriate for the database type. The fields not required for a given type are unavailable ("grayed out") in the dialog.
6. You can also change the data source name shown in the **Name** field and the database type shown in the **Type** drop-down list. Changing the data source name changes how it is listed in the lists in the dialog for both the **Import From Database** and **Export to Database**. Changing which fields are available may require you to specify different information depending on the database type chosen. If you change the database type, it is a good idea to change the name to identify it as a different data source in the list.
7. Click the **OK** button to accept your changes.

You can also remove direct data sources. Use caution, as you can remove the four direct data sources that are provided with Spotfire S+. If you do, you can add them again following the procedures above to add a data source. To remove a data source:

1. From the **File** menu, select the **Export Data ▶ To Database**.
2. Select the direct data source from the **Data Target** list you wish to modify.
3. Click the **Modify Target** button, and the **Modify Direct Source** dialog appears.
4. Click the **Remove** button in this dialog to remove the data source.

How Direct Data Sources are Stored

The entries in the data sources drop lists appearing in the **Import From Database** and **Export to Database** dialogs are actually stored in a special text file located in the **.Prefs** subfolder of your project folder **[S_PROJ]\.Prefs**. The file is called **datasources.ini**, and can be edited with any text editor.

Each line of the file is a comma-delimited specification of the necessary information for the data source, as in the following example:

```
DIRECTDB:Direct DB2,direct-db2,,testdb,testqa,testqa
DIRECTDB:Direct Oracle,direct-oracle,ORACLE.TESTDB,,testqa,
DIRECTDB:Direct SQL Server,direct-sql,,,,
DIRECTDB:Direct Sybase,direct-sybase,qa.insightgul.com,,,
```

Each line must begin with the string `DIRECTDB`: This allows Spotfire S+ to distinguish the information as pertaining to direct data sources. Following this string, each field which appears in the dialog is entered in the following order, with commas separating the fields:

```
[name], [type], [server name], [database name], [username],
[password]
```

where `[name]` is the data source name that appears in the drop-down lists in the dialogs. The `[type]` field must be one of the following (matching the type field in the `importData` and `exportData` commands):

- `direct-db2`
- `direct-oracle`
- `direct-sql`
- `direct-sybase`

The `[password]` field must be specified as clear text.

Unspecified or blank field values must be separated by commas, as in the example below where the database name and password fields are left unspecified:

```
DIRECTDB:Direct Oracle,direct-oracle,ORACLE.TESTDB,,testqa,
```

As an alternative to managing direct database sources in the dialogs, you can simply edit this file after closing Spotfire S+. Restart Spotfire S+ so that the changes you made to the data sources in this file are used in Spotfire S+.

INSTALL AND CONFIGURE DATABASE CLIENTS (UNIX)

In testing direct driver support on Linux® and UNIX® platforms, we have found that the database vendors provide fairly complete installation instructions regarding database clients on supported platforms.

Please refer to the installation instructions that came with the database software to install database clients on your system.

INSTALL AND CONFIGURE DATABASE CLIENTS (WINDOWS)

In order to use direct database driver support in Spotfire S+ for Windows, you must install database client software on the same system where Spotfire S+ is installed. Currently, Spotfire S+ supports 32-bit versions of the following database clients:

- SQL Server 2000
- Sybase 12.5
- Oracle 9i
- DB2 7.2

The SQL Server 2000 client supports access to a variety of SQL Server versions, including SQL Server 6.5. The Oracle 9i client supports access to most previous versions of Oracle, including 8i.

Other versions of the database clients may work but have not been tested by TIBCO Software Inc..

You can follow the instructions provided by a database vendor to install a database client. Alternatively, you can follow the steps below for the appropriate client. These instructions are provided to help get you started quickly installing and using a particular database client. For more in-depth information, consult the instructions provided with the database client software.

SQL Server 2000 Client

1. Insert the Microsoft SQL Server 2000 Enterprise Edition CD in your CD-ROM drive and run **English ► Ent ► autorun.exe**.
2. In the dialog that appears, select **SQL Server 2000 Components** from the available setup options.
3. In the next screen, select to install the **Database Server**.
4. In the next screen, select **local computer**, and select a destination on your system.
5. In the next screen, select **Create a new installation of SQL Server, or Install Client Tools** from the available options.

6. In the next screen, choose **Client Tools Only** from the available options
7. In the feature installation screen, accept all the selected features – *do not remove any*!
8. Once installed, run the **Enterprise Manager** from the **Start** menu using the icon at **Programs ▶ Microsoft SQL Server ▶ Enterprise Manager**.
9. In the **SQL Server Enterprise Manager** window, there is a sub-window called **Console Root ▶ Microsoft SQL Servers**. In the left pane of this window, expand the tree view to **Console Root ▶ Microsoft SQL Servers ▶ SQL Server Group**.
10. Right-click this expanded node and select **New SQL Server Registration** from the menu.
11. In the **Select a SQL Server** dialog which appears, choose the name of the server system you installed SQL Server on from the list on the left, and click the **Add** button to add it to the right.
12. In the dialog **Select an Authentication Mode**, choose the radio button for **SQL Server Authentication**.
13. Next, specify the appropriate login name and password when prompted.
14. Finally, choose to add the SQL server to the existing group called **SQL Server Group**.

After you have successfully setup the client following the steps above, ensure that the paths below are in your PATH environment variable:

```
[install path]\80\Tools\BINN
```

where `[install path]` is the path you chose to install the client tools in step 4 above.

Sybase Client

Insert the Sybase 12.5 Adaptive Enterprise client CD into your CD-ROM drive. The installation program should automatically start.

Note

If the Sybase installer crashes or locks up during startup in **java.exe**, then you may need to disable Java "just-in-time" compiling on your system. See Sybase technical article **www.sybase.com/detail/1,6904,1013241,00.html** on the Sybase Web site for more information about this problem.

Carry out the following steps to install a Sybase client on your system:

1. Choose the **Standard** install.
2. Accept the default location or specify another location. Make sure that the install directory you choose does *not* contain any spaces.
3. Ensure that the following paths are in your `PATH` environment value:

```
[install path]\CFG-1_0\bin;[install path]\OCS-
12_5\dll;[install path]\OCS-12_5\lib3p;[install
path]\OCS-12_5\bin
```

where `[install path]` is the path you chose in step 2 above.

After installation, select **Programs ▶ Sybase ▶ dsedit** from the **Start** menu to start this utility program. Then do the following steps:

1. Click **OK** on the first screen to open the **Interfaces Driver** screen.
2. From the **Server Object** menu, select **Add**.
3. In the **Input Server Name** box, enter the network internet protocol name of the server running Sybase, and click **OK** (e.g., **qaimage.tibco.com**).
4. In the **attributes** column of the **Interfaces Driver** dialog, double-click the server address row.
5. In the **Network Address Attribute** window, click the **Add** button.
6. Select **TCP** as the network connection protocol from the drop-down list and enter

[server ip name], 2048

in the **Edit** field. **[server ip name]** is the internet protocol name of the server that has Sybase installed, and should be the same as specified in step 3 above, as in **qaimage.tibco.com**. 2048 is the port number it receives on, and you may have to change the port number, depending on the server. Click **OK** to accept the changes.

The **[server ip name]** you specify here is used in Spotfire S+ to connect to the server and use Sybase.

7. Click **OK** to accept the **Network Address Attribute** window.
8. Test the connection to the new server by selecting **Ping** from the **Server Object** menu.
9. In the **Ping** dialog, click the **Ping** button. You should see another dialog appear, indicating that the connection was successful.
10. Close the **dsedit** utility program.
11. Test the connection using the Sybase ISQL utility:
    - Open a DOS Command window and type the following:

      ```
      isql –Utestqa –Ptestqa – S[server ip name]
      ```
    - At the isql program prompt, enter the following:

      ```
      select * from pubs2.dbo.sales
      go
      ```

    After you type **go**, you should get an output table printed in the window.

Oracle Client

Insert the Oracle 9i Client for Win32 CD in your CD-ROM drive. The setup starts automatically, and you can follow these steps:

1. In the **Oracle Universal Installer: File Locations** dialog, enter the path to install the Oracle client software on your system in the **Destination path** field.
2. For the type of installation, select **Administrator**.
3. Use the default port number of 2030.

4. After installation, the **Oracle Net Configuration Assistant** automatically appears. If it does not automatically appear, you can manually start it from the **Start** menu at **Programs ▶ Oracle - OraHome90 ▶ Configuration and Migration Tools ▶ Net Configuration Assistant**.

5. In the **Net Configuration Assistant Wizard**, enter the values specified in the appropriate wizard steps indicated below:

   - Select **Perform typical configuration** in the **Welcome** step. Click **Next**.
   - Select **No, I will create net service names myself...**. Click **Next**.
   - Select **Oracle8i or later database or service**. Click **Next**.
   - Specify an appropriate network service name. It is suggested that you specify a name that contains the name of the database and your network domain, as in **testdb.tibco.com** for the service name. Click **Next**.
   - Select **TCP** as the protocol. Click **Next**.
   - Specify the name of the server which is running Oracle on your network. Specify only the system name (and not an IP address), such as **qadb-s2k** for the host name, and specify an appropriate port number. You can accept **1521** as the default port or change it, depending on how your server is configured. Click **Next**.
   - Select **Yes, perform a test** to test the connection. Click **Next**. You should get an unsuccessful connection screen with a **Change Login** button on it.
   - Click the **Change Login** button, and specify the appropriate username and password in the dialog. Click **OK** and then **Next**.

6. Now, the test connection should report success. Click **Next**.

7. Accept or change the network service name shown. This is the name you specify in Spotfire S+ to access the ORACLE server. Click **Next**.

8. When asked whether you want to configure another net service name, choose **No**. Click **Next**.

After a successful installation, ensure that the following paths are in your PATH environment value:

```
[install path]\ora90\bin
```

where `[install path]` is the path you chose during setup in step 1 above.

DB2 Client

Insert the DB2 Universal Database Enterprise Edition Version 7.2 CD into your CD-ROM drive. The installer should start automatically.

1. Choose **Install** from the list of setup options.
2. In the **Select Products** dialog, choose **DB2 Administration Client**.
3. Select **Typical** setup type.
4. Select a destination for the installation or accept the default path.
5. In the **Enter Username and Password for Control Center Server** dialog, enter your network login name and password. Note that you can enter another username and password, but using your network login and password make it easy to remember.
6. You may receive a dialog telling you that you don't have privileges to do certain things with DB2 on your system. Click the **OK** button in this dialog to continue with the setup.
7. If you are prompted to restart your system, make sure you do this. Some services that are installed as part of the DB2 installation need to be installed and mounted.
8. After your system is restarted or after the end of a successful setup, the **First Steps** dialog appears. In this dialog, select **Catalog Sample Databases** from the list on the left.
9. The **Client Configuration Assistant** is started.
10. In the **Welcome** dialog, click the **Add Database** button.

11. A "wizard" dialog appears. In the **Source** page, click the **Search the network** radio button, and switch to the **Database name** page.
12. In the **Tree** view, expand the **Other Systems** node and wait until the program has scanned all network systems for DB2 servers. The list it finds appears below this node.
13. Look for the name of the server that has DB2 installed on it in this tree. If you don't see it, check the server, and make sure DB2 has been properly started. Once it is located, expand the server name node in the tree to see a list of databases.
14. Select the database you want to use from the **Local databases** node. The name appears in the **Target Database** field at the bottom of the dialog. Click **Next**.
15. In the **Alias** page, verify that the alias for this database is listed as the database name you chose. Click **Next**.
16. In the **ODBC** page, accept the defaults for registering the database as a system source. Click **Finish**.

    Click the **Close** button when a confirmation dialog appears. Do not attempt to test the connection at this time as the settings are not correct yet. You are then returned to the **Client Configuration Assistant** main window.
17. You can see that the database you selected was added as a database. Select this database, then click the **Properties** button. This opens another dialog called **Database Properties – [database name]**.
18. Click the **Properties** button. This opens another dialog called **Update Connection Wizard – [database name]**.
19. Protocol should be set to **TCP/IP**. Click **Next**.
20. Correct the host name by changing it to the IP name of the database server, such as **qadb-snt.tibco.com**. Click the **Finish** button. You are returned to the **Database Properties – [database name]** dialog. Click **OK**.
21. In the main **Client Configuration Assistant** window, select the database you selected from the list and click the **Test** button.

22. In the **Connect to DB2 Database**, enter the appropriate username and password in the fields and leave other default settings. Click the **OK** button. You should receive a "connection successful" dialog.
23. Close the **Client Configuration Assistant**.
24. Close the **First Steps** dialog.
25. Ensure that the path `[install path]\BIN` is in your `PATH` environment value. `[install path]` is the path you selected in step 4 above.

DEBUGGING YOUR FUNCTIONS

11

INTRODUCTION

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 Spotfire S+, it is often more efficient to simply type in a smaller function, then test it and see what improvements it might need.

Spotfire S+ provides several built-in tools for debugging your functions. In general, these tools make use of the techniques described in Chapter 7, Writing Functions in Spotfire S+, to provide you with as much information as possible about the state of the evaluation.

In this chapter, we describe several techniques for debugging S-PLUS functions using these built-in tools as well as the techniques of Chapter 3, Computing on the Language, to extend these tools even further. For a discussion of debugging loaded code, see Chapter 8, Interfacing with C and FORTRAN Code, in the *Application Developer's Guide.* Refer also to Chapter 2, Data Management, for a detailed discussion of frames.

BASIC SPOTFIRE S+ DEBUGGING

When an error occurs in a S-PLUS expression, Spotfire S+ generally returns an error message and the word `Dumped`:

```
> acf(corn.rain,type="normal")
Problem in switch(itype + 1,: desired type of ACF is
unknown
Use traceback() to see the call stack
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 Spotfire S+ 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()
6: eval(action, sys.parent())
5: doErrorAction("Problem in switch(itype + 1,: desired
 type of ACF is unknown",
4: stop("desired type of ACF is unknown")
3: acf(corn.rain, type = "normal")
2: eval(expression(acf(corn.rain, type = "normal")))
1:
Message: Problem in switch(itype + 1,: desired type of
ACF is unknown
```

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 `data`*n*, 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)
```

Spotfire S+ 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 Spotfire S+ output by using `print` or `show` instead (the `show` function was introduced in Spotfire S+ 5.0 as a more object-oriented version of `print`):

```
> 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.)

Using recover

The `recover` function can be used to provide interactive debugging as an error action. To use `recover`, set your error action as follows:

```
options(error=expression(if(interactive())
    recover() else dump.calls()))
```

Then, for those type of errors which would normally result in the message "Problem in ... Dumped," you are instead asked "`Debug? Y/N`"; if you answer "`Y`", you are put into `recover`'s interactive debugger, with a `R>` prompt. Type `?` at the `R>` prompt to see the available commands. Use `up` to move up the frame list, `down` to move down the list. As you move to each frame, `recover` provides you with a list of local variables. Just type the local variable name to see its current value. For example, here is a brief session that follows a faulty call to the `sqrt` function:

```
> sqrt(exp)

Problem in x^0.5: needed atomic data, got an object of class
"function"
```

```
Debug ?  ( y|n ): y
Browsing in frame of x^0.5
Local Variables: .Generic, .Signature, e1, e2

R> ?
Type any expression. Special commands:
`up', `down' for navigation between frames.
`where' # where are we in the function calls?
`dump'  # dump frames, end this task
`q'     # end this task, no dump
`go'    # retry the expression, with corrections made
Browsing in frame of x^0.5
Local Variables: .Generic, .Signature, e1, e2
R> up
Browsing in frame of sqrt(exp)
Local Variables: x
R(sqrt)> x
function(x)
.Internal(exp(x), "do_math", T, 108)
R(sqrt)> x<-exp(1)
R(sqrt)> go
[1] 1.648721
```

In the example session, we accidentally gave a function as the argument to `sqrt`, rather than the needed atomic data object. Inside `recover`, we move up to `sqrt`'s frame, change the argument `x` to the result of a function *call*, then use `recover`'s `go` command to complete the expression.

INTERACTIVE DEBUGGING

Although `print`, `show`, 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.

Using `recover` provides interactive debugging, but it has no real debugging facilities–the ability to step through code a line at a time, set breakpoints, track functions, and so on.

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 `inspect`.

Starting the Inspector

To start a session with the inspector, call `inspect` 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 `inspect` as follows:

```
> inspect(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 `inspect` 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 11.1. 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 `help` instruction to see a list of instructions; type help *instruction* for `help` on a particular instruction.

To leave the inspector and return to the Spotfire S+ prompt, use the instruction `quit`.

Examining Variables

You can obtain a listing of objects in the current evaluation frame with the inspector 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 inspector 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 Spotfire S+ 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 the examples `fcn.C` and `fcn.D` given in Matching Names and Values on page 39 of the *Application Developer’s Guide*. 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 Entering, Marking, and Tracking Functions on page 378 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 11.1: *Instructions for the interactive inspector.*

Keyword	**Help given**
`help [ `*`instruction`*` | names | keywords ]`	Provides help on *instruction*, `names`, or `keywords`. With no arguments, `help` gives a summary of the available instructions.
`complete [loop | function]`	Evaluates to the end of the next for/while/repeat loop, or to the point of function return.
`debug.options [echo = T|F] [marks = hard|soft]`	With `echo=T`, expressions are printed before they are evaluated. With `marks=hard`, evaluation always halts at a marked expression. With `marks=soft` it halts only during a resume. Setting `marks=soft` is a way of temporarily hiding marks for `do`, `complete`, etc. The defaults are: `echo=F`, `marks=hard`. With no arguments, `debug.options` displays the current settings.
`do [`*`n`*`]`	Evaluates the next *n* 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, `do` does the entire group.
`down [`*`n`*`]`	Changes the local frame for instructions such as `objects` and `eval` to be *n* frames deeper than the current one. The default is 1. After any movement of the evaluator (`step`, `resume`, etc.), the local frame at the next stop is that of the function stopped in.
`enter`	Enters the function called in the next expression.

Table 11.1: *Instructions for the interactive inspector.*

Keyword	Help given
`eval` *`expr`*	Evaluates the S-PLUS expression *expr*.
`find` *`name`*	Reports where *name* would be found by the evaluator.
`fundef [`*`name`*`]`	Prints the original function definition for *name*. Default is the current function. Tracked and marked functions will have modified function definitions temporarily installed; `fundef` is used to view the original. The modified and original versions will behave the same; the modified copy just incorporates tracing code.
`mark`	Remembers the current expression; evaluation will halt here from now on.
`mark` *`name1`* `[`*`name2`* `...] [at entry|exit]`	Arranges to stop in the named functions. The default is to stop at both entry and exit.
`objects`	Names of objects in this function's frame.
`on.exit`	Displays the current on-exit expressions for this function.
`quit`	Abandons evaluation, return to top-level prompt.
`resume`	Resumes evaluation.
`return.value`	Displays the return value, if known.
`show [tracks | marks | all]`	Displays installed tracks and marks. Default all.
`step [`*`n`*`]`	Evaluates the next *n* expressions. Default 1.
`track` *`name1/`* `[`*`name2/`* `... ] [at entry|exit] [print = T|F] [with expr]`	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 `expr`.
`unmark` *`name1/`* `[`*`name2`* `...] [at entry|exit]`	Deletes mark points at the named locations in the named functions.
`unmark` *`n1`* `[`*`n2`* `...]`	Deletes mark points *n1*, *n2*, ... See mark and show.
`unmark all`	Deletes all mark points.

Table 11.1: *Instructions for the interactive inspector.*

Keyword	Help given
`untrack name1/ [name2/ ... ]`	Disables tracking for the named functions.
`up [n]`	Changes the local frame for instructions such as `objects` and `eval` to be *n* frames higher than the current one. The default is 1. After any movement of the evaluator (`step`, `resume`, etc.), the local frame at the next stop is that of the function stopped in.
`where`	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 sub-expression. 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 Entering, Marking, and Tracking Functions on page 378. 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:

```
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 begin 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 7, Writing Functions in Spotfire S+. 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(
     sapply(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("ElTime", get("StopTime", frame=1) -
      get("StartTime", frame=1), frame=1)
  cat(deparse(substitute(fun)), "took time",
      get("ElTime", 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 7, Writing Functions in Spotfire S+. 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> quit
> 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> eval smallp <- 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`.
- 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 Spotfire S+ debugging needs. On occasion, however, you may find some of Spotfire S+'s other debugging tools of some use. This section briefly describes these other tools—`browser`, `debugger`, and `trace`.

Using the Spotfire S+ 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(make.data.sets)>` 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(make.data.sets)>
```

Type `?` at the prompt to get brief help on the browser, plus a listing of the variables in the local frame:

```
b(make.data.sets)> ?
Type any expression. Special commands:
`up', `down' for navigation between frames.
`c'     # exit from the browser & continue
`stop'  # stop this whole task
```

```
`where' # where are we in the function calls?
Browsing in frame of make.data.sets(5)
Local Variables: n, names
b(make.data.sets)> names
[1] "data 1" "data 2" "data 3" "data 4" "data 5"
```

To leave the browser, type either `c` or `q` at the prompt:

```
b(make.data.sets)> q
>
```

You can type arbitrary S-PLUS expressions at the `browser` prompt. These expressions are evaluated in the chosen frame, which is indicated by the function name within the prompt–thus, `b(make.data.sets)>` indicates that you are in `browser` in the frame of the function `make.data.sets`. Thus, you can type alternative expressions and see if a possible fix will actually work.

Using the Spotfire S+ 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 (`?`, `up`, `down`, and frame numbers) to move through the dumped frames.

For example, consider the following simple function:

```
debug.test <-
function()
{
        x <- 1:10
        sin(z)
}
```

This has an obvious error in the second line of the body, so it will fail if run. To use `debugger` on this function, do the following:

```
> options(error=expression(dump.frames()))
> debug.test()
Problem in debug.test(): Object "z" not found
Evaluation frames saved in object "last.dump", use
debugger() to examine them
> debugger()
Message: Problem in debug.test(): Object "z" not found
browser: Frame 11
b(sin)>
```

You are now in the browser, and can view the information in the dumped frames as described above.

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 Spotfire S+ 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.frames` 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)>
```

OBJECT-ORIENTED PROGRAMMING IN SPOTFIRE S+ 12

INTRODUCTION

Throughout the first chapters, almost no mention has been made of object-oriented programming. Yet one of the very first statements in this book was that Spotfire S+ 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 `show(`*object*`)`, you expect Spotfire S+ 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 `show` function would need to be modified every time a new class of objects was created. In object-oriented programming, however, the `show` 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 Spotfire S+ 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("White","Black","Gray","Gray","White","White")
> yyy <- factor(xxx)
> show(xxx)
[1] "White" "Black" "Gray" "Gray" "White" "White"
> show(yyy)
[1] White Black Gray Gray White White
```

The distinct look of the printed factor arises because factors are a distinct class of object, with their own `show` method.

This chapter describes the essentials of the Spotfire S+ object-oriented language.

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 `show` 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 `show(`*myobject*`)` with *myobject* a matrix, Spotfire S+ will print *myobject* using the matrix method for `show`. If *myobject* is a numeric vector, the printing is performed by `show`'s default method.

As a Spotfire S+ 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* Spotfire S+'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.

Classes and Methods in Spotfire S+

Spotfire S+ looks for methods according to *signatures*, which can be either a single character string specifying the name of a class or a named list matching the classes to the formal arguments of the function. If Spotfire S+ finds no method for the most specific signature for the given arguments, it looks in turn at each of the signatures which might apply. As soon as Spotfire S+ 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 build objects of a specific class, you generally define a *constructor*, or *generator*, function. Typically, generator functions have the name of the object they create—`matrix`, `numeric`, and so on.

Generator functions are not strictly necessary; Spotfire S+ includes the function `new` to allow you to generate new objects of any class. Typically, however, you will embed a call to `new` within your generator function.

You can view the class of any object with the `class` function:

```
> class(state.x77)
[1] "matrix"
```

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 Spotfire S+ software; any time you use the Spotfire S+ 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.

Prototype and Representation

A *prototype*, in the context of this chapter, is the basic template used to create an instance of a classed object. For objects with slots, the default prototype is normally specified by the class's *representation*, which assigns each named slot to a particular class. If the slot is assigned a virtual class, such as `"vector"`, you must also provide a prototype for that slot, because you can not instantiate an object with a virtual class. You may also want to provide a prototype for a slot with a regular class if you'd like the default object to be something other than the default object of the corresponding class. For example, the `.Dim` slot of the `matrix` class needs to have length 2, instead of the length 0 of the default `integer` object. This can be specified by providing a prototype.

Inheritance and Extension; Is Relations

Whenever a class is created from an existing class in such a way that all existing methods for the existing class continue to work for objects of the new class, we say that the new class *extends* the existing class, and that the new class *inherits* the methods of the existing class, or simply, that the new class inherits from the existing class. In S-PLUS 5.0 and later, inheritance is much more rigorous than it was in earlier versions of Spotfire S+; you can no longer define arbitrary inheritance structures and expect them to work.

Is relations allow you to test and specify inheritance relations. For example, some arrays are matrices, but only those which have a length 2 `.Dim` slot. We can formalize this by defining an `Is` relationship using `setIs` as follows:

```
setIs("array", "matrix",
     test = function(object)length(dim(object))==2)
```

Metadata

Generic functions, methods, and class definitions are all stored in ordinary S-PLUS objects, but these objects are not stored on ordinary databases. Instead, they are stored, with mangled names, in meta databases that accompany each Spotfire S+ chapter. The idea is that information about the class system and its operation is not really data, it is information *about* the data, and thus it makes sense to separate it from the actual data stored in the ordinary databases.

You can view and manipulate objects in meta databases using the standard functions `objects`, `get`, `exists`, and so, by specifying `meta=1` as one of the arguments. For example, we can list the objects in the working database's meta database as follows:

```
> objects( meta=1)
 [1] "C#circle"    "C#point"     "C#rectangle"
 [4] "C#zseven"    "Classes"     "Generics"
 [7] "Groups"      "M#Ops"       "M#[<-"
[10] "M#[<-<-"     "M#coerce"    "M#draw"
[13] "M#jitter"    "M#show"
```

For most purposes, however, you will want to manipulate these objects using special functions written to manage the metadata. Throughout this chapter we will use these special functions, such as `setMethod`, `setClass`, etc.

DEFINING NEW CLASSES IN SPOTFIRE S+

Defining new classes in Spotfire S+ involves specifying precisely what information defines the class, sharply contrasting objects within the class from those outside the class. For example, a matrix is defined to be a structure with an integer vector of length 2 specifying the number of rows and columns, with an optional set of row and column names. Objects that don't satisfy the defining requirements are not members of the class.

As an example of new classes in Spotfire S+, 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 (the *anchor* for the figure) and a point on the circle. A square is specified by one anchor corner and a side length, while a rectangle is specified by one anchor corner and its opposite diagonal corner. These specifications provide the necessary information for defining our new classes.

To actually define the class, we can use the function `setClass` with either a *representation* defining named slots or a *prototype* that specifies what default objects can be combined to create a default object of the new class. For our purposes, named slots seem useful, so we want to specify a representation, as in the definition of the class `"point"`:

```
> setClass("point", representation(x="numeric",
          y="numeric"))
```

We then use this fundamental `point` class as a central part of our representation of our shapes:

```
> setClass("circle", representation(center="point",
          radius="numeric"))
> setClass("rectangle", representation(anchor="point",
          diagonal="point"))
```

Whenever you specify a class definition with `setClass`, Spotfire S+ creates an object of class `classRepresentation` in the corresponding meta database. This object, which can be viewed by calling the function `getClass`, contains the information specified by the call to setClass as well as information about the class that might be provided after its original definition by calls to functions such as setIs.

We can view the information about our circle class as follows:

```
> getClass("circle", complete=F)

Slots:
  center    radius
 "point" "numeric"
```

We also need the following utility function, as.point, in the examples which follow:

```
as.point <-
function(p)
{
  if(is.numeric(p) && length(p)==2)
      list(x=p[1], y=p[2])
  else if(is.list(p) && !is.null(p$x) && !is.null(p$y))
      p
  else if(is.matrix(p))
      list(x=p[,1], y=p[,2])
  else stop("Cannot interpret input as point")
}
```

Defining Generator Functions

Our motivation for defining these shapes is to create a rudimentary drawing tool 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* (or *constructor*) function for circles:

```
circle <-
function(center, radius, point.on.edge)
{
    center <- as.point(center)
    val <- NULL
    if(length(center@x) == 2) {
        val <- new("circle", center = new("point",
                        x = center@x[1], y = center@y[1]),
                        radius = sqrt(diff(center@x)^2 +
                        diff(center@y)^2))
    }
```

```
        else if(length(center@x) == 1) {
            if(missing(radius)) {
                point.on.edge <- as.point(point.on.edge)
            }
            else if(is.atomic(radius)) {
                 val <- new("circle", center = center,
                            radius = abs(radius))
            }
            else {
                point.on.edge <- as.point(radius)
            }
            if(is.null(val)) {
                 val <- new("circle", center = new("point",
                           x = center@x[1], y = center@y[1]),
                           radius = sqrt((point.on.edge@x -
                           center@x)^2 + (point.on.edge@y -
                           center@y)^2))
            }
        }
        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 Spotfire S+ command line:

```
> simple.circle <- circle(center = c(0.5, 0.5),radius=0.25)
> simple.circle
An object of class "circle"

Slot "center":
An object of class "point"

Slot "x":
[1] 0.5

Slot "y":
[1] 0.5


Slot "radius":
```

```
[1] 0.25
```

Note the recursive nature of our representation for circles. Objects of class `circle` have two slots, one of which is occupied by a `point` object which also contains two slots.

Defining Methods

The default printing for circles seems rather too formal and unnecessarily tied to the formal representation of the object, when all we really need to see is a center and radius. Thus, it makes sense to define a method for use with the `show` generic function. To define a method, you use the `setMethod` function, which in its simplest form takes three arguments: a character string specifying the generic function to which the method applies, a character string specifying the signature (typically, just the class) for which the method applies, and the actual method definition.

Here is our definition:

```
> setMethod("show", "circle",
function(object)
{
     cat(" Center: x = ", object@center@x, "\n ",
      " y =", object@center@y, "\n ",
      "Radius:", object@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 must ensure that its arguments match those of the generic.

You can specify the function definition in either of two ways. The first, which we used in the definition of our `show` method for circles, puts a function definition (or, equivalently, the name of an ordinary function object) in the method; this definition is then stored in the meta database as the definition for the method. The second, which you may find preferable if you've worked with Spotfire S+ for a long time, puts a function *call* in as the definition of the method. This allows you to define an ordinary function on an ordinary database as your basic method definition, and then have the actual S-PLUS method stored on the meta data call this function. There are, however, some drawbacks to this second approach. In particular, if your function needs to use `substitute`, `sys.parent`, or similar functions, the function call method will not work because the function call is evaluated in frame 2, not the top-level frame 1.

Defining Generic Functions

You create generic functions with the function `setGeneric`. Generic functions in Spotfire S+ tend to be extremely simple, thanks to the utility function `standardGeneric`. The `standardGeneric` is the standard body of a generic function, which simply indicates that the generic just dispatches a method to do the real work. The typical generic function consists of a single call to `standardGeneric.` For example, 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:

```
> setGeneric("draw", function(x, ...)
          standardGeneric("draw"))
```

This standard definition 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`. In our case, there is no default method for `draw`. Here, for example, is a `circle` method for the generic function `draw`:

```
> setMethod("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 Spotfire S+ evaluator finds the appropriate method and draws a circle on the current graphics device.

For ordinary functions that you would like to make generic, you need not create the generic explicitly. Simply define a method for a non-default class, and Spotfire S+ automatically creates a generic function and takes the existing ordinary function and turns it into the new generic's default method.

As an illustrative example, consider the `jitter` function, used to separate points for plotting. What `jitter` does is add a small amount of noise to each observation, which enables points to be distinguished without altering the actual shape of the data very much. You can make `jitter` generic in your home directory by defining a character method for it; the character method pastes together the original character vector and a random number printed to `factor` significant digits:

```
> isGeneric("jitter")
[1] F
> setMethod("jitter", "character", function(x, factor=1){
        paste(x, format(runif(length(x)), digits=factor),
        sep=".")
        }
)
redefining function "jitter" to be a generic function on
database ".Data"
Warning messages:
  Conflicting definitions of "jitter" on databases ".Data"
and  "splus" in: assign(f, what@genericDef, where = where)
> jitter(state.name)
 [1] "Alabama.0.8 "         "Alaska.0.9 "
 [3] "Arizona.1   "         "Arkansas.0.2 "
 [5] "California.0.3 "      "Colorado.0.3 "
 [7] "Connecticut.0.4 "     "Delaware.0.4 "
 . . .
> isGeneric(jitter)
[1] T
```

Warning

Note that although the `jitter` function is now generic, it is generic *only* on the working data, not in the system databases. In particular, if you are interested in defining a system function as generic for everyone at your site, you will need to modify your site's **.S.init** file to include a system-wide directory in which the function has been defined to be generic.

EDITING METHODS

Because methods aren't stored on ordinary databases with ordinary names, you can't simply edit them with `fix` as you would ordinary functions. Instead, you must dump them, using the `dumpMethod` function, edit them with your favorite text editor, such as the Spotfire S+ **Script** window in Microsoft Windows®, then source the file back in to SPOTFIRE S+.

To dump a method with `dumpMethod`, you need to specify the generic function name and the appropriate signature for the method you want to edit. For example, if we want to edit our character method for `jitter`, we can use `dumpMethod` as follows:

```
> dumpMethod("jitter", "character")
```

The file **jitter.character.q** is created when `dumpMethod` is called. Opening the file with a text editor or **Script** window (in Windows) shows the following:

```
setMethod("jitter", "character",
function(x, factor = 1)
{
        paste(x, format(runif(length(x)), digits =
                factor), sep = ".")
}
)
```

Note that the output is in standard dump format, suitable for use by `source`.

If you want to edit all the methods for a given generic, you can dump all the methods at once using the `dumpMethods` function. This can sometimes be dangerous, however, particularly if you have methods spread over several libraries. In general, it is safest to edit just one method at a time.

GROUP METHODS

Four groups of S-PLUS functions, all defined as calls to `.Internal` or `.Call`, 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 (for example, `sin`, `exp`); the *Math2* group, containing just two functions, `round` and `signif`, that are like the functions in the Math group but take an additional argument; 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; note that *Arith*, *Compare*, and *Logic* are all groups as well as *Ops*..

Table 12.1: *Functions affected by group methods*

Group	Functions in Group

Ops	`Arith: "+" (unary and infix), "-" (unary and infix), "*", "^", "%%", "%/%", "/"`
	`Compare: "==", ">", "<", "!=", "<=", ">=", compare`
	`Logic: "!"(unary not), "&", "|"`

Math	`abs, acos, acosh, asin, asinh, atan, atanh, ceiling, cos, cosh, cumsum, cumprod, exp, floor, gamma, lgamma, log, log10, sin, sinh, sqrt, tan, tanh, trunc`
Math2	`round, signif`
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 24 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 are defined in the same way as ordinary methods, using `setMethod`. If the method handles all the functions in the group in the same way, it can be quite simple, as for example in the `Summary` method for class `numericSequence`:

```
> getMethod("Summary", "numericSequence")
function(x, ..., na.rm = F)
callGeneric(as(x, "numeric"), ..., na.rm = na.rm)
```

Caution

One caution about the Summary group–it does not include either `mean` or `median`, both of which are implemented as Spotfire S+ 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 nontrivial group method, we will define a group of operators for the finite field Z7. This field consists of the elements {*a7* = 0, *b7* = 1, *c7* = 2, *d7* = 3, *e7* = 4, *f7* = 5, *g7* = 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, `c7 * e7 = b7 = 1`, `d7 / f7 = c7 = 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 *Z7* as follows:

```
> setClass("zseven", prototype=integer(0))
```

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
  x <- new("zseven", x)
  x
}
```

The following example shows the value returned by a typical input vector:

```
> zseven(c(5,10,15))
An object of class "zseven"

[1] 5 3 1
```

We can suppress the printing of the class information by defining a method for `show`:

```
> setMethod("show", "zseven", function(object)
{
  object <- unclass(object)
  print(object)
}
```

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`, with `e2` defaulting to `NULL` to match the generic:

```
> setMethod("Ops", "zseven", function(e1,e2=NULL) {})
```

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 Z7 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 Spotfire S+'s internal code. This passing is accomplished via the function `callGeneric`, which determines the name of the currently called function and constructs a call to it with the arguments provided (we'll define the `inverse` function later in the chapter):

```
        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 <- callGeneric(e1, e2)
```

Finally, we need to insure that numeric results are of class `zseven`, while logical results are passed back unchanged:

```
  switch(class(value),
           integer = zseven(value),
           logical = value)
```

Put together, the complete method looks like this:

```
  > setMethod("Ops", "zseven",
  function(e1, e2=NULL) {
    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 <- callGeneric(e1, e2)
      }
      switch(class(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:

```
   setMethod("/", "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.

Note on Inherited Classes

S searches for a matching method for a generic function using the following algorithm:

First, it looks for a method that has an exact match of the signature.

Second, it looks for a group generic method with an exact match of the signature.

Third, it searches for methods that match using inherited classes.

It searches for inherited-class methods starting with the last argument, and stops as soon as it finds a method for the function or group generic that matches a set of actual and inherited classes. Therefore, it will preferentially match a signature with the exact class for the first argument, and an inherited class for the second argument, over another method that might have an exact match for the second argument, and inherited class for the first argument.

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 <- integer(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.

EXTRACTION AND REPLACEMENT METHODS

Extraction functions are functions that extract subsets of objects to create new objects. The most common extraction function in S-PLUS is "[", the subset operator. If your new class extends an old class in a natural way (as our zseven class extends the integers), you may not need to define extraction methods. If your new class is built from components of different classes with competing extraction methods, you will probably want to create extraction methods to make subset operations well-defined.

Suppose, for example, that we define a class of *ordered pairs* with a representation as two numeric vectors. We can do this as follows:

```
> setClass("opair", representation(x="numeric",
          y="numeric"))
```

We can create a generator function for this class as follows:

```
> opair <- function(x,y){ new("opair", x=x, y=y) }
```

If we then create an instance of an ordered pair object and try to extract its first element, we get an error:

```
mypair <- opair(x=1:10,y=(1:10)^2)
mypair[1]
Problem in mypair[1]: function "[" not defined for
non-vector class "opair"
Use traceback() to see the call stack
```

To remedy this, we need to define an extraction method. A simple one is easily defined–it takes advantage of each of the slots being a numeric vector and uses the existing vector methods within the slots:

```
> setMethod("[", "opair", function(x,..., drop=F)
           opair(x@x[...], x@y[...]))
> mypair[1]
An object of class "opair"

Slot "x":
[1] 1

Slot "y":
[1] 1
```

When you define a method, you must be careful to make sure the method's arguments are the same as the generic's. You can use the function `functionArgNames` to quickly verify the generic's argument names:

```
> functionArgNames("[")
[1] "x"    "..."   "drop"
```

Replacement functions are functions that can appear on the left side of an assignment arrow, typically replacing either an element or attribute of their arguments. 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:

```
setReplaceMethod("[", "zseven",
function(x, ..., value)
{
  if (is.complex(value) || value %% 1 != 0)
      stop("Replacement not meaningful for this value")
  x <- unclass(x)
  x[...] <- value %% 7
  zseven(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 SPOTFIRE S+ 13

THE GUI TOOLKIT

Spotfire S+ is equipped with a graphical user interface (GUI) toolkit for programmers to create and manipulate GUI components: menus, toolbars, dialogs and graphics. The Spotfire S+ GUI toolkit is a set of S-PLUS functions that enables communications between Spotfire S+ applications and Windows. It provides facilities for the following applications:

1. Automating an interactive Spotfire S+ session.
2. Extending or customizing the existing Spotfire S+ GUI.
3. Developing a new GUI on top of Spotfire S+.

This toolkit is object-oriented and can operate on virtually any Spotfire S+ GUI object. The functionality of the GUI toolkit is also available using the point-and-click operations in dialogs. The toolkit approach is for Spotfire S+ 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 Spotfire S+ program, or from **Script** windows, which open when a script file is opened.

A set of sample script files, shipped with Spotfire S+, illustrate various uses of the toolkit (these are located in the **splus/samples/dialogs** directory). An example showing how to create and display a simple function dialog is listed in Table 13.1.

Note

This chapter applies to Windows users only. The `guiCreate` and `guiModify` functions (and associated functions) are not available in the UNIX® version.

Table 13.1: *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
#-------
guiCreate("Property", Name = "simple1Prop0", DialogControl = "String",
          DialogPrompt = "MyReturn", DefaultValue = "w");
guiCreate("Property", Name = "simple1Prop1", DialogControl = "String",
          DialogPrompt = "&Y Value", DefaultValue = "30");
#-------
# Step 3: create the function info object
#-------
guiCreate("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.
#-------

guiDisplayDialog("Function", Name= "simple1");
```

Running this script file displays the dialog in Figure 13.1. There are two ways to run a script, either from the menus by opening the script file, then clicking the **Run** toolbar button, or from entering the following command to the **Commands** window:

```
> source("simple1.ssc")
```

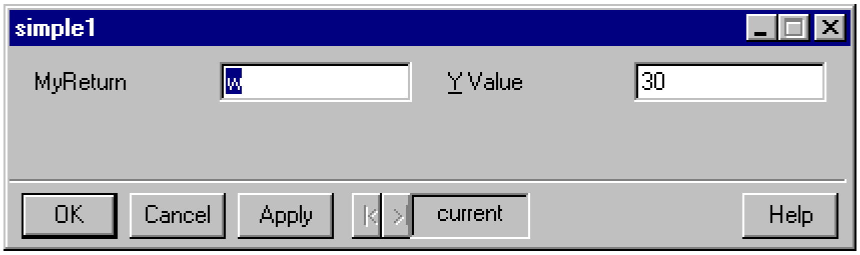


Figure 13.1: *The dialog created by* ***simple1.ssc***.

GUI Objects

GUI objects are the building blocks of the graphical user interface subsystem of Spotfire S+. 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 Spotfire S+ 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:

```
> guiGetClassNames()
```

GUI objects created are listed in the Object Explorer, but are not stored in the standard Spotfire S+ 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: `guiCreate`, `guiModify`.

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 Spotfire S+ 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 a Spotfire S+ 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 as the end of a session.

guiCreate

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 `guiGetArgumentNames(classname)` to return a list containing the valid argument names for the class in question.

Note

In the current version of `guiCreate`, creating an object with the same class and object name as an existing one would modify the existing object. You must use `guiRemove` before `guiCreate` to ensure the clean creation of a new object.

Table 13.2: *Arguments to* `guiCreate`.

Argument	**Required**	**Description**
`classname`	Required	A character string specifying the class of the object to be created. Use `guiGetClassNames()` to get a list of all class names which can be used here. There are many options, including `"Property"`, `"BoxPlot"`, `"XAxisTitle"`, and so on.
`Name`	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 `guiGetArgumentNames(classname)` 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 graphsheet name “`GS1`”, then the function to create this box plot would be

```
> guiCreate("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 `GS1`, you would specify:

```
> guiModify("MainTitle",Name = "$$GS1$1$1", Title ="Title")
```

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,

```
> guiCreate("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,

```
> guiCreate("Arrow", Name = "$$GRAPH1$ARROW1")
```

This command will create the arrow named `ARROW1` in the graph `GS1` in the current graph sheet.

```
> guiCreate("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

```
> guiCreate ("Property", Name = "simple1Prop1",
+ DialogControl = "Integer")
```

This will create a property object called `simple1Prop1`. The Object Explorer can be used to examine the contents of this object.

See Also

guiCopy, guiModify, guiOpen, guiOpenView, guiSave, guiGetArgumentNames, guiGetClassNames, guiGetPropertyValue, guiDisplayDialog

guiCopy

This function 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 `guiCreate` for more details on the implications of this.

Table 13.3: *Arguments to* `guiCopy`.

Argument	Required	Description
`classname`	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.

Examples

```
> guiCopy ("Property", Name="simple1Prop1",
+ NewName="simple1Prop2")
```

```
> guiCopy ("Property", Name="simple1Prop1",
+ NewName="simple1Prop2", DialogControl="Float",
+ DefaultValue="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 `guiGetPropertyValue` function.

See Also

`guiCreate`, `guiModify`, `guiOpen`, `guiOpenView`, `guiSave`, `guiGetArgumentNames`, `guiGetClassNames`, `guiGetPropertyValue`, `guiDisplayDialog`

guiModify

This function modifies 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 13.4: *Arguments to* `guiModify`.

Argument	**Required**	**Description**
`classname`	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

```
> guiModify("Property", Name = "simple1Prop1",
+ DialogControl = "String", DefaultValue = "OK")
```

This will modify the `simple1Prop1` property object to use `String` as its dialog control type, with the value “`OK`”.

See Also

`guiCreate`, `guiCopy`, `guiModify`, `guiOpen`, `guiOpenView`, `guiSave`, `guiGetArgumentNames`, `guiGetClassNames`, `guiGetPropertyValue`, `guiDisplayDialog`

guiMove

This function 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 `guiCreate` for more details.

Table 13.5: *Arguments to `guiMove`.*

Argument	Required	Description
`classname`	Required	A character string specifying the class of an object to be moved.
`Name`	Required	A character string specifying the object path name.
`NewName`	Required	A character string specifying the destination object path name.
Optional arguments allow an object to be modified while being moved.		

Example

```
> guiMove ("Property", Name="simple1Prop1",
+ NewName="simple1Prop3", DefaultValue = "good")
```

This will move the `simple1Prop1` property object to `simple1Prop3`. Also, the `DefaultValue` property is modified to “`good`.” The Object Explorer can be used to examine the contents of this object.

See Also

guiCreate, guiCopy, guiModify, guiOpen, guiOpenView, guiSave, guiGetArgumentNames, guiGetClassNames, guiGetPropertyValue, guiDisplayDialog

guiOpen

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 = FullScreen. This function is a property command, see guiCreate for more details.

Table 13.6: *Arguments to* guiOpen.

Argument	Required	Description
docClassname	Required	A character string specifying the class of a document object to be opened: Script, GraphSheet, Report, or ObjectBrowser.
FileName	Required	A character string giving the name of the file including the file's title plus the entire directory path.
Hide	Optional	FALSE will open and display the object, TRUE will open the window but it will not become the active window.
Show	Optional	One of: "Normal", "Minimized", "Maximized", "FullScreen". FullScreen only applies to GraphSheet.

Table 13.6: *Arguments to* `guiOpen`.

Argument	Required	Description
`Duration`	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.
`Top` `Left`	Optional	"Auto" lets the system decide on the window size, or specific coordinates can be specified for the top-left corner of the window.
`Width`	Optional	"Auto" or a specific width
`Height`	Optional	"Auto" or a specific height

Example

```
> guiOpen( "Script", FileName = "C:\\Program
Files\\TIBCO\\splus82\samples\\dialogs\\simple1.ssc")
```

This will open the script file specified by `FileName` and display it in a script window.

See Also

`guiCreate`, `guiCopy`, `guiModify`, `guiOpen`, `guiOpenView`, `guiSave`, `guiGetArgumentNames`, `guiGetClassNames`, `guiGetPropertyValue`, `guiDisplayDialog`

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 `guiOpen`.

Example

```
> guiOpenView("data.frame", Name = "car.all")
```

This will open a grid view for a data frame called `car.all`.

See Also

```
guiCreate, guiCopy, guiModify, guiOpen, guiOpenView,
guiSave, guiGetArgumentNames, guiGetClassNames,
guiGetPropertyValue, guiDisplayDialog
```

guiRemove

Remove the object identified by `Name` and `classname`.

Table 13.7: *Arguments to* `guiRemove`*.*

Argument	Required	Description
`classname`	Required	A character string specifying the class of an object to be removed.
`Name`	Required	A character string specifying the object path name.

.

Example

```
> guiRemove("Property", Name = "simple1Prop3")
```

This will delete the property object `simple1Prop3`. This object should disappear from the Object Explorer listing.

See Also

```
guiCreate, guiCopy, guiModify, guiOpen, guiOpenView,
guiSave, guiGetArgumentNames, guiGetClassNames,
guiGetPropertyValue, guiDisplayDialog
```

guiSave

Saves the document identified by `Name` and `docClassname` to the file specified by `FileName`.

Table 13.8: *Arguments to* `guiSave`.

Argument	Required	Description
`docClassname`	Required	A character string specifying the class of an document object to be saved, from the same range of options as `guiOpen`.
`Name`	Required	A character string specifying the source object path name.
`FileName`	Required	A character string giving the name of the destination path and file.

Example

```
> guiSave( "Script", Name = "$$simple1",
+ FileName = "C:\\work\\examples\\dialogs\\simple1.ssc")
```

This will save the script document `simple1` as:

```
"C:\\work\\guilocal\\examples\\dialogs\\simple1.ssc"
```

on your hard drive.

See Also

`guiCreate`, `guiCopy`, `guiModify`, `guiOpen`, `guiOpenView`, `guiGetArgumentNames`, `guiGetClassNames`, `guiGetPropertyValue`, `guiDisplayDialog`

guiRemoveContents

Use `guiRemoveContents` to remove the objects contained by the specified container.

For example,

```
> guiRemoveContents("GraphSheet", Name=guiGetGSName)
```

will clear the contents of the current graph sheet, leaving it blank.

guiSetRedraw(Name, Redraw)

This function allows you to control when a **Graph Sheet** is redrawn after modifications are made to it by calls to `guiModify()`.

Table 13.9: *Arguments to* `guiSetRedraw`

Argument	Required	Description
`Name`	Required	A character string specifying the **Graph Sheet** name.
`Redraw`	Not required	Determines whether **Graph Sheet** automatically redraws whenever there are modifications made to it. Default = T.

Example

Create a new Graph Sheet with a line plot:

```
> guiCreate( "GraphSheet", Name="GS1" )
> guiCreate( "LinePlot", Name = "GS1$1$1",
     DataSet = "fuel.frame",
     xColumn = "Weight",
     yColumn = "Disp.",
     LineStyle = "Solid" )
```

Turn off automatic updating of the Graph Sheet while we change the attributes of the line plot:

```
> oldRedraw <- guiSetRedraw(Name="GS1", Redraw=F)
```

Modify the line plot; the Graph Sheet will not redraw:

```
> guiModify("LinePlot", Name="GS1$1$1", LineColor="Lt
     Green")
```

Restore the previous redraw setting; the Graph Sheet now redraws:

```
> guiSetRedraw(Name="GS1", Redraw=oldRedraw)
```

See Also

`guiModify`

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,
guiGetPropertyValue, 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:      Name
        Default:     ""
     DataSet
        Prompt:      Data Set
        Default:     ""
     xColumn
        Prompt:      x Columns
        Default:     ""
     yColumn
        Prompt:      y Columns
        Default:     ""
     zColumn
        Prompt:      z Columns
        Default:     ""
     wColumn
        Prompt:      w Columns
        Default:     ""
     PlotConditionType
        Prompt:      Type
        Default:     "Auto"
        Option List: [ Auto, None, Specified Columns ]
```

Spotfire S+ 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

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 13.10: *Arguments to* `guiGetArgumentNames`.

Argument	**Required**	**Description**
`classname`	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] "Name"                      "Type"
 [3] "DefaultValue"              "ParentProperty"
 [5] "DialogPrompt"              "DialogControl"
 [7] "ControlProgId"             "ControlServerPathName"
 [9] "Range"                     "OptionList"
[11] "PropertyList"              "CopyFrom"
[13] "OptionListDelimiter"       "HelpString"
[15] "SavePathName"              "IsRequired"
[17] "UseQuotes"                 "NoQuotes"
[19] "IsList"                    "NoFunctionArg"
[21] "Disable"                   "IsReadOnly"
[23] "NoStripSpaces"
```

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

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 13.11: *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", "simple1Prop1")
 [1] ""      "Normal"  ""        ""      ""        "Integer"
 [7] ""      ""        ""        ""      ""        ""
[13] ""      ""        ""        "F"     "F"       "F"
[19] "F"     "F"       "F"       "F"     "F"
```

```
> guiGetPropertyValue("Property", "simple1Prop1",
  PropName="Type")
[1] "Normal"
```

See Also

`guiCreate, guiCopy, guiModify, guiOpen, guiOpenView, guiSave, guiGetArgumentNames, guiGetClassNames, guiGetPropertyValue, guiDisplayDialog`

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", "BorderStyle")
 [1] "None"          "Solid"        "Dots"          "Dot Dash"
 [5] "Short Dash"    "Long Dash"    "Dot Dot Dash"  "Alt Dash"
 [9] "Med Dash"      "Tiny 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 `BorderStyle` property information is as follows:

```
> guiGetPropertyPrompt("Box", "BorderStyle")
$PropName:
[1] "BorderStyle"

$prompt:
[1] "Style"
```

```
$default:
[1] "Solid"

$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

This function displays a dialog associated with a GUI object. The dialog can be optionally displayed as *modal* or *modeless*, using the `bModal` argument.

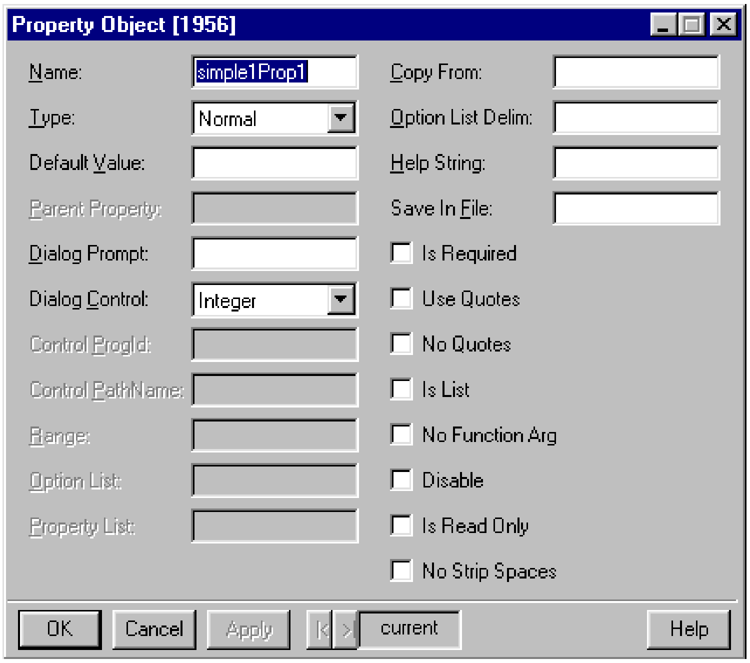
Table 13.12: *Arguments to* `guiDisplayDialog`.

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.
`bModal`	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 13.2.



Property Object [1956]

Name:	simple1Prop1	Copy From:	
Type:	Normal	Option List Delim:	
Default Value:		Help String:	
Parent Property:		Save In File:	
Dialog Prompt:		☐ Is Required	
Dialog Control:	Integer	☐ Use Quotes	
Control ProgId:		☐ No Quotes	
Control PathName:		☐ Is List	
Range:		☐ No Function Arg	
Option List:		☐ Disable	
Property List:		☐ Is Read Only	
		☐ No Strip Spaces	

Figure 13.2: *The* `simple1Prop1` *property object is created after running the* ***simple1.ssc*** *script.*

See Also

`guiCreate`, `guiCopy`, `guiModify`, `guiOpen`, `guiOpenView`, `guiSave`, `guiGetArgumentNames`, `guiGetClassNames`, `guiGetPropertyValue`

guiModifyDialog

This function is used to modify the current value of a live active dialog. It enables communications between two or more active dialogs.

Table 13.13: *Arguments to* `guiModifyDialog`.

Argument	Required	Description
`wndID`	Required	The Window (Dialog) ID of a live active dialog, obtained through a 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`.

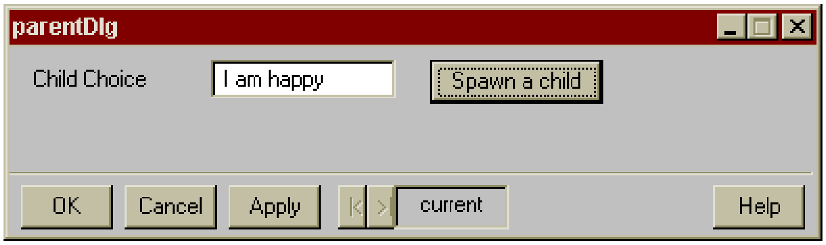
Example

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 13.14: *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 13.14 dialogs should be displayed as shown in Figure 13.3.


parentDlg

Child Choice: I am happy

Figure 13.3: *Click on* ***Spawn a child****, then select* ***happy****, to create the child dialog shown.*

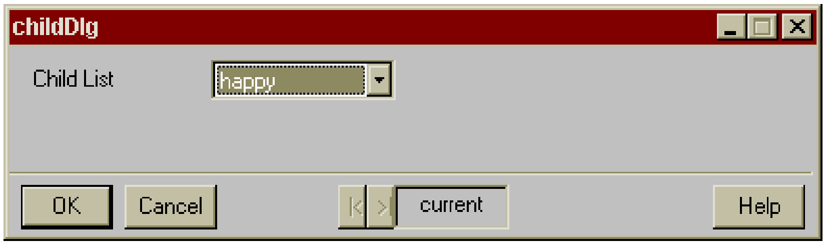


Figure 13.3: *Click on* ***Spawn a child****, then select* ***happy****, to create the child dialog shown.*

See Also

`guiDisplayDialog`, `guiCreate`, `guiCopy`, `guiModify`, `guiOpen`, `guiOpenView`, `guiSave`, `guiGetArgumentNames`, `guiGetClassNames`, `guiGetPropertyValue`

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. Spotfire S+ provides programmatic access to determine what objects are selected.

guiGetSelectionNames

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 13.15: *Arguments to* `guiGetSelectionNames`.

Argument	Required	Description
`classname`	Required	A character string specifying the class of the object in question.
`ContainerName`	Optional	The container object name, for example: "`$$DS1`" (datasheet 1) or "`$$GS1`" (graphsheet 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 in the current graphsheet.

Table 13.15: *Arguments to `guiGetSelectionNames`.*

Argument	Required	Description
`StartSelection`	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.
`EndSelection`	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

```
> guiGetSelectionNames("factor")
> guiGetSelectionNames("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.

guiSetRowSelection

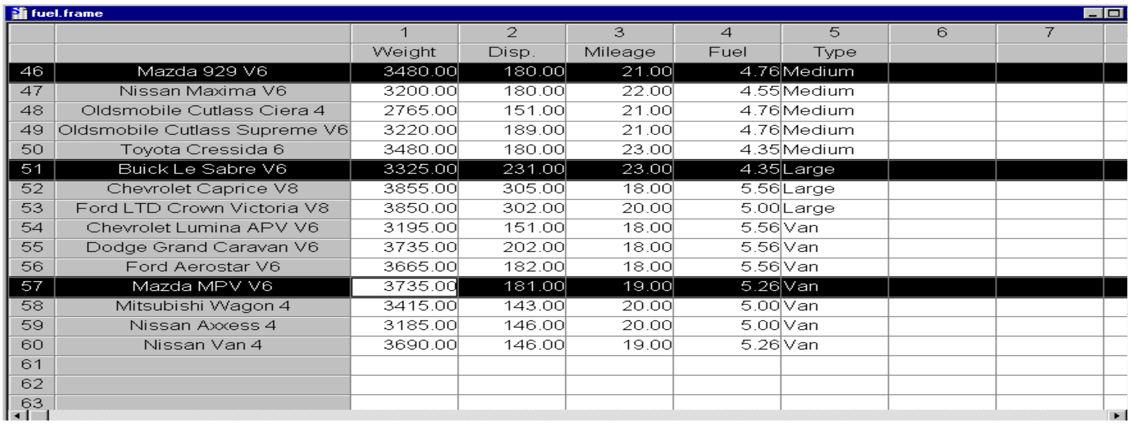
Use the `guiSetRowSelection` one or more rows of a data set as "selected"; that is, they appear highlighted in a **Data** window view, and plotted symbols appear highlighted in a **Graph** window. This selection can be done interactively in the GUI; this function permits the same behavior programmatically. This is useful, for example, if you want to highlight known outliers in a data set.

guiGetRowSelection

Use the `guiGetRowSelection` function to obtain a list of rows in the current data set that are selected.

guiGetRowSelectionExpr

Use the `guiGetRowSelectionExpr` function to obtain a 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 13.4.



		1	2	3	4	5	6	7
		Weight	Disp.	Mileage	Fuel	Type		
46	Mazda 929 V6	3480.00	180.00	21.00	4.76	Medium		
47	Nissan Maxima V6	3200.00	180.00	22.00	4.55	Medium		
48	Oldsmobile Cutlass Ciera 4	2765.00	151.00	21.00	4.76	Medium		
49	Oldsmobile Cutlass Supreme V6	3220.00	189.00	21.00	4.76	Medium		
50	Toyota Cressida 6	3480.00	180.00	23.00	4.35	Medium		
51	Buick Le Sabre V6	3325.00	231.00	23.00	4.35	Large		
52	Chevrolet Caprice V8	3855.00	305.00	18.00	5.56	Large		
53	Ford LTD Crown Victoria V8	3850.00	302.00	20.00	5.00	Large		
54	Chevrolet Lumina APV V6	3195.00	151.00	18.00	5.56	Van		
55	Dodge Grand Caravan V6	3735.00	202.00	18.00	5.56	Van		
56	Ford Aerostar V6	3665.00	182.00	18.00	5.56	Van		
57	Mazda MPV V6	3735.00	181.00	19.00	5.26	Van		
58	Mitsubishi Wagon 4	3415.00	143.00	20.00	5.00	Van		
59	Nissan Axxess 4	3185.00	146.00	20.00	5.00	Van		
60	Nissan Van 4	3690.00	146.00	19.00	5.26	Van		
61								
62								
63								

Figure 13.4: ***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 `guiGetRowSelectionExpr` as follows:

```
> guiGetRowSelectionExpr("fuel.frame")
[1] "46,51"
```

You can select those same rows in a later session using `guiSetRowSelection`:

```
> guiSetRowSelection("fuel.frame", "46,51")
```

OPTIONS

All elements of the Spotfire S+ interface are under programmatic 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 design 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`:

- `guiGetAxisLabelsName`: returns the name of the `AxisLabels` for a specified axis (axis 1 by default).
- `guiGetAxisName`: returns the name of the axis for a specified axis (axis 1 by default).
- `guiGetAxisTitleName`: returns the name of the axis title for a specified axis (axis 1 by default).

- `guiGetGSName`: returns the name of the current `GraphSheet`. (This function takes no arguments.)
- `guiGetGraphName`: returns the `GraphName` of the graph with the specified `GraphNum` in the specified `GraphSheet`.
- `guiGetPlotName`: returns the `Name` of the plot with the specified `PlotNum` in the specified `GraphNum` in the specified `GraphSheet`.

For example,

```
> guiPlot("Line"DataSetValues=data.frame(1:20, sin(1:20))
> guiModify("YAxisTitle", Name=guiGetAxisTitleName(),
  Title="sin(x)")
```

guiGetPlotClass

Use the `guiGetPlotClass` 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 `guiModify`.
2. If no plot type is specified, return a list of valid plot types. These are the valid plot types for `guiPlot`.

For example,

```
> guiGetPlotClass("Scatter")
[1] "LinePlot"
> guiGetPlotClass()
 [1] "Scatter"                 "Isolated Points"
 [3] "Bubble"                  "Color"
 [5] "Bubble Color"            "Text as Symbols"
 [7] "Line"                    "Line Scatter"
 [9] "Y Series Lines"          "X Y Pair Lines"
[11] "Y Zero Density"          "Horiz Density"
[13] "Robust LTS"              "Loess"
[15] "Spline"                  "Supersmooth"
[17] "Horiz Step"              "Dot"
[19] "Kernel"                  "Vert Step"
[21] "High Density"            "Robust MM"
...
```

```
> guiPlot("Loess", DataSetValues=environmental[,1:2])
> guiModify(guiGetPlotClass("Loess"), Name=
+ guiGetPlotName(), LineColor="Red")
```

guiUpdatePlots To update the plots created by `guiPlot(DataSetValues=...)` with new data set values, use `guiUpdatePlots`.

For example,

```
> gsName <- guiPlot("Scatter", DataSetValues=fuel.frame
  [,1:2])
> guiUpdatePlots(GraphSheet=gsName, DataSetValues=
  environmental[,1:2])
```

The number of columns in the data set used in `guiUpdatePlots` should be the same as the number of columns in the original data set used in `guiPlot`.

UTILITIES

Utility functions are available to perform GUI actions not related to specific objects.

guiRefreshMemory

Use the `guiRefreshMemory` 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 `guiExecuteBuiltIn` 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 `guiGetPropertyValue` function. For example, suppose we wanted to view the **About Spotfire S+** 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:

```
> guiExecuteBuiltIn("$$SPlusMenuBar$Object_Browser$Window$
Tile-Vertical")
```

We can then use this command in a call to `guiExecuteBuiltIn`:

```
> guiExecuteBuiltIn(
"$$SPlusMenuBar$Object_Browser$Help$About_S_PLUS")
```

SUMMARY OF GUI TOOLKIT FUNCTIONS

Table 13.16: *Summary of graphics interface functions.*

Function	Description
`guiCreate`	Creates all interface objects.
`guiCopy`	Copies one object to another, possibly with modifications.
`guiModify`	Modifies an object.
`guiMove`	Moves an object, possibly with modifications.
`guiOpen`	Opens an object of type **Script**, **GraphSheet**, **ObjectBrowser** or **Report**.
`guiOpenView`	Opens an object that is viewable by the Object Explorer.
`guiRemove`	Deletes an object created by `guiCreate`.
`guiSave`	Saves an object to a file.
`guiGetClassNames`	Lists all available object types.
`guiGetPropertyValue`	Lists details of property values for a given object.
`guiGetArgumentNames`	Lists all relevant arguments for a given object.
`guiDisplayDialog`	Displays the identified dialog.

Table 13.16: *Summary of graphics interface functions.*

Function	Description
`guiModifyDialog`	Modifies a currently active dialog, and can be used in conjuction with callback functions.
`guiGetSelectionNames`	Reports the objects currently selected by the user.

USING LESS TIME AND MEMORY 14

INTRODUCTION

In Chapter 7, Writing Functions in Spotfire S+, 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 Spotfire S+ allocates and manages memory. This information is important for understanding why, for example, for loops are less efficient in Spotfire S+ 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.

Spotfire S+ makes extensive use of virtual memory. To your Spotfire S+ session, your total memory is the sum of your main memory and swap space. Spotfire S+ 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 Spotfire S+ 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 Spotfire S+ Allocates Memory

To write the most efficient S-PLUS functions, you should have some understanding of how Spotfire S+ allocates and frees memory. Such understanding takes you a long way toward understanding why loops and recursion are discouraged in Spotfire S+ programming.

In Chapter 2, 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 Spotfire S+ corresponds to memory allocation for the various frames. At a lower level, memory is allocated by *arenas* and *buckets*.

There are two types of arenas, *standard* and *custom*. Most data in S-PLUS objects are stored in custom arenas, with the exception of character data, which may be stored in standard arenas. Data and headers stored in custom arenas are *reference counted*, that is, a record is kept of which frames are using a given object, and the arena cannot be freed until the reference count reaches zero, that is, when no frame is using the object. Atomic data objects consist of the data plus a 40 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. Some scratch space not in S-PLUS objects may be in standard arenas; such arenas may be shared with other data.

The vector headers for all objects in a given frame are stored in buckets. Each bucket can hold up to 75 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 Spotfire S+ is finished with the frame, values are returned to the parent frame. If the value occupies a custom arena, the arena is simply reassigned to the parent frame. Once any necessary values are transferred to the parent frame, the frame is broken down and all its associated memory is freed. Any standard arenas associated with the frame are destroyed at the same time.

Why and When Spotfire S+ Copies Data

Spotfire S+ 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 a 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 Spotfire S+.) In order to ensure that a function cannot alter data in its caller's frame, Spotfire S+ must increment reference counts for arguments to functions. It does not copy arguments. Spotfire S+ copies data only when the objects are modified. In particular, if a data set is named, its value cannot be changed by arbitrary functions, so it will be copied. Spotfire S+ 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, Spotfire S+ 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 character data may be stored in a standard arena, Spotfire S+ does copy content of that memory to an arena of the caller's frame. Thus, if your function returns a large object consisting of character data it may 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. S-PLUS 5.x and later makes significantly fewer copies of data sets than earlier versions of Spotfire S+; this is largely a result of fewer copies needing to be made thanks to the existence of reference counting.

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 a Spotfire S+ 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 a S-PLUS function. It reports the maximum amount of memory used in Spotfire S+ 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`. To get a baseline memory usage, call `memory.size()` as the first function in a session, then quit Spotfire S+, start it up again, and call your function followed by `memory.size()` in the same top-level expression.

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:

- Start a new Spotfire S+ session. When you get the Spotfire S+ prompt, call `memory.size` and quit:

```
> memory.size()
[1] 536576
> q()
```

- Note the baseline memory returned by `memory.size`.
- Start a second Spotfire S+ session. When you get the Spotfire S+ prompt, create a braced expression containing your function call and a call to `memory.size`:

```
> {f(); memory.size()}
[1] 2543616
```

```
> q()
```

- Call `mem.tally.reset()`:

```
> mem.tally.reset()
```

- When the Spotfire S+ prompt returns, create a braced expression containing your function call and a call to `mem.tally.report`:

```
> {f();mem.tally.report()[2]}
 evaluation
    2004460
```

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`.

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
  tmp2 <- trunc(tmp1)
  mean(tmp2 > 0.5)
}
```

This requires 4.5 million bytes to complete, while the following slightly modified version needs only 2.5 million:

```
g1 <- function(n = 125000)
{
  tmp <- runif(n)
  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 Spotfire S+ 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 is to call the function `storageSummary`:

```
storageSummary <- function(frame = sys.parent(),
                           print.zeros = F)
{
        if(is.loaded("S_table_header_types"))
                .C("S_table_header_types",
                        frame,
                        print.zeros)
        s <- storage()
        headers <- sum((s$headers - s$freed)[s$
                "frame (h)" == frame])
        arenaBytes <- sum(s$used[s$frame == frame])
        cat(headers, "headers in use and", arenaBytes,
                " bytes of arena storage in use in frame",
                frame, "\n")
        invisible(list(headers = headers, arenaBytes
                 = arenaBytes))
}
```

The storageSummary function is normally 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.)

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()
2 headers in use and 192 bytes of arena storage in use in
frame 2
7 headers in use and 1000244 bytes of arena storage in use
in frame 2
424 headers in use and 2005112 bytes of arena storage in use
in frame 2
502 headers in use and 3006028 bytes of arena storage in use
in frame 2
934 headers in use and 3010388 bytes of arena storage in use
in frame 2
$headers:
[1] 934

$arenaBytes:
[1] 3010388

> g1()
2 headers in use and 192 bytes of arena storage in use in
frame 2
7 headers in use and 1000208 bytes of arena storage in use
in frame 2
8 headers in use and 1000208 bytes of arena storage in use
in frame 2
9 headers in use and 1000208 bytes of arena storage in use
in frame 2
10 headers in use and 1000208 bytes of arena storage in use
in frame 2
$headers:
[1] 10

$arenaBytes:
[1] 1000208
```

Start Spotfire S+:

```
> g()
nframe=2 bytes=20064 frame size=154 memory.size=536576
nframe=2 bytes=1020064 frame size=1000240
memory.size=1540096
nframe=2 bytes=2020064 frame size=2000281
memory.size=3547136
nframe=2 bytes=3020064 frame size=3000322
memory.size=5554176
nframe=2 bytes=4020064 frame size=3000322
memory.size=6557696
NULL

> q()
```

Start Spotfire S+ again:

```
> g1()
nframe=2 bytes=20064 frame size=154 memory.size=536576
nframe=2 bytes=1020064 frame size=1000240
memory.size=1540096
nframe=2 bytes=1020064 frame size=1000240
memory.size=3547136
nframe=2 bytes=1020064 frame size=1000240
memory.size=3547136
nframe=2 bytes=2020064 frame size=1000240
memory.size=4550656
NULL

> q()
```

WRITING GOOD CODE

Use Vectorized Arithmetic

Spotfire S+ 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 Spotfire S+:

```
> 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. Spotfire S+ 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 `NA`'s in the data. If your data have no missing values, you can safely use logical subscripts.)

Avoid for Loops

In Chapter 7, Writing Functions in Spotfire S+, 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.

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 Spotfire S+. 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 Spotfire S+ 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*-1st 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" atomic data sets, either in loops or in recursive function calls. Spotfire S+ maintains each atomic data object in a contiguous portion of memory. If the data object grows, it may outgrow the available contiguous memory allotted to it, requiring Spotfire S+ 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])
 . . .
```

Spotfire S+ 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 Spotfire S+ 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 Constructing Return Values on page 212 in Chapter 7, Writing Functions in Spotfire S+.

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. Spotfire S+ 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 12, Object-Oriented Programming in Spotfire S+, 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 Spotfire S+ 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 Spotfire S+ 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
}
```

Using Non-Generic Functions

If you know ahead of time which method you are going to be using, call that function in your loop instead of the generic. The overhead of generic dispatch to select the appropriate method for a given object will cause the call to the generic to run hundreds of times slower.

You can significantly increase the speed in a loop such as `apply`, `for`, or `while` by using a non-generic function. The following shows the dramatic improvement in performance between the use of a generic and non-generic method for the function `max`:

```
> unlist(lapply(c("max","min"), isGeneric))
[1] T T
> unlist(lapply(c("all", "any", "max","min", "prod","sum"),
isGeneric))
[1] T T T T T T
> showMethods("max")
      Database                   x
[1,] "splus"  "ANY"
[2,] "splus"  "positionsCalendar"
[3,] "splus"  "timeSpan"
> sys.time({for(i in 1:10000)max(1,2)})
[1] 54.609 55.049
```

Get a function from the method `max`:

```
> maxDefault <- selectMethod("max") #get the default method
> maxDefault
function(x, ..., na.rm = F)
.Internal(max(x, ..., na.rm = na.rm), "do_summary", T, 114)
> sys.time({for(i in 1:10000)maxDefault(1,2)})
[1] 0.230 0.231
```

Comparing the two methods in `sys.time`, we get

```
> 54/.23
[1] 234.7826
```

In this case, using a non-generic method for `max` is almost 235 times faster!

IMPROVING SPEED

By default, Spotfire S+ now checks to see whether your system is an Intel Pentium processor, and if so, uses Intel's Math Kernel Library BLAS routines. These routines are optimized for Intel Pentiums and thus significant speed-up should be observed in certain Spotfire S+ operations (such as matrix multiplication) that call BLAS routines. Significant speed-up of certain operations can be obtained when using a Pentium multi-processor machine. The operations for which Spotfire S+ can take advantage of the additional processors are those (such as matrix multiplication) in which the BLAS routines of the Intel Math Kernel Library are used. See `intelmkl.use` for more information.

Using these routines on a non-Intel Pentium processor may cause some problems. It is also possible that the check Spotfire S+ performs to detect an Intel processor may currently be detecting a Pentium in all cases, even when your system has a non-Intel processor. Spotfire S+ includes a few S language functions to allow you to control whether Intel's BLAS routines or S-PLUS's BLAS routines are used:

- `is.intelmkl.inuse()` returns a logical indicating whether the BLAS routines used are from Intel's Math Kernel Library (if FALSE, the BLAS routines used are from the S-PLUS engine).
- `intelmkl.use(set = T, number.of.processors = 1)` allows you to change which set of BLAS routines are used. (To use the Spotfire S+ engine BLAS, use `set=F`.) If Intel's Math Kernel Library BLAS routines are to be used (`set=T`), `number.of.processors` allows you to specify how many processors of a multi-processor machine should be used (if not specified, any previous specification remains in effect; the default is 1).

- `intelmkl.processor.count()` returns an integer specifying how many processors of a multi-processor machine are used when Intel's Math Kernel Library BLAS routines are to be used. This number should never be larger than the number of processors on the machine being used.

Warning

If you are using a non-Intel processor, Windows may erroneously report to Spotfire S+ that you are using a Pentium processor and cause Spotfire S+ to use the Intel Math Kernel Library BLAS routines.

If you are using a non-Intel processor and you encounter problems with any Spotfire S+ operations, try setting the environment variable

```
S_USE_INTELMKL=no
```

on the Spotfire S+ start-up command line.

SIMULATIONS IN SPOTFIRE S+ 15

INTRODUCTION

In Chapter 14, Using Less Time and Memory, we describe how you can employ knowledge of Spotfire S+ computations to write functions that use time and memory more efficiently than those you might otherwise write. The main message of Chapter 14 is to use vectorized S-PLUS functions to do as much as possible with each function call.

In this chapter, we consider some special problems that arise in writing large simulations with Spotfire S+. Here, we are interested in cases where calculations cannot be vectorized, either because of their complexity or because the vectors are too large to fit into virtual memory. Specifically, we show different approaches to dealing with the following problems:

1. Working with many data sets in a loop.
2. Iterating a large number of times (>50,000) in a loop.
3. Predicting the amount of time required for a simulation and monitoring its progress.
4. Recovering after errors.

WORKING WITH MANY DATA SETS

Spotfire S+ uses a *caching* technique to increase the speed of most ordinary computations:

- When Spotfire S+ reads 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 Spotfire S+ sees a reference to the data set, it uses the stored version so that it does not waste time reading the disk again.
- When asked to save a permanent data set, Spotfire S+ stores the data until the end of the top-level expression before writing it to disk. Although Spotfire S+ may alter the data many times while evaluating the expression, it takes the time to write it to disk only once. Typically, writing to disk is about 1,000 times slower than writing to main memory.

While this caching is a good trade-off between memory and speed for most Spotfire S+ purposes, it can use excessive memory when reading or writing many large permanent data sets in a loop. Each of the data sets is stored in main memory until the end of the current top-level expression, which consumes memory that can otherwise be dedicated to computations.

WORKING WITH MANY ITERATIONS

A loop in the S-PLUS language tends to slow after many iterations. This is not usually noticeable until about 10,000 iterations, and not terribly important until about 50,000 iterations. The slowdown occurs because functions invoked by "quick calls," such as arithmetic, comparison, and subscripting functions, leave behind a 32-byte chunk of memory with each call. The memory is not freed until the top-level function completes. The Spotfire S+ memory compaction mechanism spends time testing to see if it can free the chunks; as they build up, the mechanism spends most of its time dealing with those chunks that will never be freed. Thus, asymptotically, the time it takes to evaluate a `for` loop is quadratic in the number of iterations, where the quadratic factor has a small coefficient that you cannot detect until about 10,000 iterations. In this section, we discuss two functions that delay the inherent slowdown in large loops: `lapply` and `For`.

The Advantages of lapply

As we mention in the section Avoid for Loops (page 468), loops are more efficient if they simply call a function, rather than calling each line of the function individually. Thus, you can delay the slowdown inherent in large loops by replacing the body of a loop with a function that accomplishes the same thing. For example, replace the lines of code

```
for(i in 1:n)
{
  #
  # some lines of code
  #
  results[i] <- final.result
}
```

with:

```
f <- function(<arguments>)
{
  #
  # some lines of code
  #
  return(final.result)
}
```

```
for(i in 1:n)
  results[i] <- f(<argument values>)
```

If you cannot do this, however, use `lapply` instead of an explicit `for` loop. The `lapply` function is currently the most efficient way to do looping in Spotfire S+. It applies a specified function `FUN` to all components of a list `X`:

```
> args(lapply)
function(X, FUN, ...)
```

The `lapply` function performs looping for you by calling `for`, but it makes some special memory considerations based on the results of `FUN`.

Because lists are generalized vectors, `lapply` also works on vectors of numeric values or character strings. In particular, it works when the argument `X` is a vector of indices; this is the key to replacing loops with `lapply` statements in Spotfire S+. When `X` is a vector of indices, the function `FUN` must accept an index such as `i` as its first argument. For example, the following code shows a `for`-loop approach to adding the elements in two vectors.

```
> x <- 1:10
> y <- 11:20

# Initialize z as an empty vector the same length as x & y.
> z <- vector("numeric", length(x))
> for (i in 1:length(x))
+    z[i] <- x[i] + y[i]
> z
 [1] 12 14 16 18 20 22 24 26 28 30
```

The following call to `lapply` accomplishes the same task:

```
> z <- lapply(1:length(x), function(i) x[i] + y[i])
> z

[[1]]:
[1] 12

[[2]]:
[1] 14

[[3]]:
```

```
[1] 16

[[4]]:
[1] 18

[[5]]:
[1] 20

[[6]]:
[1] 22

[[7]]:
[1] 24

[[8]]:
[1] 26

[[9]]:
[1] 28

[[10]]:
[1] 30
```

As you see from the output, `lapply` is designed to return a list. You can use the `unlist` function to return a simple vector instead:

```
> unlist(z)
 [1] 12 14 16 18 20 22 24 26 28 30
```

More generally, the code below transforms the `for` loop in the case where `x` and `y` are not stored in a permanent database such as the working directory. When `x` and `y` are local variables, you must specify arguments for them in the definition of `FUN`, and then explicitly pass them in:

```
n <- length(x)
lapply(1:n, function(i, list1, list2)
      list1[[i]] + list2[[i]],
      list1 = x, list2 = y)
```

Using the For Function

The `For` function creates a file consisting of the contents of a `for` loop. In the file, each iteration is evaluated as a separate top-level expression. This avoids both the overhead of long-running `for` loops, as well as the memory overhead that results from caching data for the duration of a top-level expression (see the section Working with Many Data Sets (page 481)). In general, the top-level `for` loop

```
for(i in 1:n)
  results[i] <- func(i)
```

gives the same results as

```
For(i=1:n, results[i] <- func(i))
```

However, the latter does not slow down as `n` becomes very large.

The `For` function evaluates its expressions in a separate Spotfire S+ session. Because of this, all data sets that `For` refers to must be permanent data sets. If you run `For` from within a function, be sure to assign the data it needs to your working directory. The `For` function also creates a permanent data set containing the current value of the index variable; in the above example, this is `i`. This permanent variable overwrites any other by the same name in your working directory.

Running each iteration in a loop as a top-level expression may save memory, but it is much slower than running a group of iterations as a single top-level expression. This is because each top-level expression spends time initializing and writing results to disk; by doing more in each expression, we can avoid some of this overhead. Thus, the `For` function has a `grain.size` argument that controls the number of iterations included in each top-level expression. If `grain.size` is too large, memory requirements increase, and if it is too small, you waste time reading and writing disk files.

A good setting for `grain.size` is such that each top-level expression takes a few minutes to evaluate. The overhead required by a top-level expression ranges from a fraction of a second to a few seconds, depending on how much data you access from disk. You can predict how long your simulation will take by running `grain.size` iterations and linearly scaling. Note that since results are saved to disk every `grain.size` iterations, you lose only the last `grain.size` results if Spotfire S+ or the computer crashes.

The `For` function also has an optional argument `first` that is useful in certain situations. The `first` argument allows you to specify an expression for Spotfire S+ to evaluate before any of the iterations.

Recall that `For` evaluates its expressions in a new session of Spotfire S+. Therefore, you may need to attach databases or start a graphics device before `For` executes; the `first` argument allows you to do this.

Hint

If the expression given to `For` is large or the number of iterations is very large, `For` itself may run out of memory while creating the command file. In addition, the command file may be too large to fit on your disk. If this is a problem, define a function from your expression or save it with mode `"expression"`. You can then use `For` to call the function or evaluate the saved expression.

MONITORING PROGRESS

Recording the Status of a Simulation

After your simulation has been running for a while, you may want to know how far it has gotten. However, you cannot safely interrupt its progress, examine the status, and resume execution. Instead, you should include code that periodically records the status of the simulation in a file. By writing to a file rather than a Spotfire S+ data set, the information is written to disk immediately and you can view the simulation's progress without using Spotfire S+. In addition, appending text to a large file is quicker than reading a large Spotfire S+ data set, adding to it, and then writing it to disk again.

The status information you choose to record should include the iteration number, a summary of results for each iteration, and enough information to restart the simulation if Spotfire S+ or the computer should crash while it is running (see the section Recovery After Errors (page 488)). You can use `options("error")` or `on.exit` to write a message in the status file when something goes wrong. For example:

```
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("Error in iteration", i, "\n ",
        file = logfile, append = T))
  for (i in 1:n)
  {
     x.i <- get(paste("x", i, sep = "."), where = 1)
     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))
     # Cancel the error report.
     on.exit()
  return(result)
}
```

If you run `analyze.data` with the default value `n=10` in a directory that contains only `x.1` and `x.2`, you receive the following error:

```
> analyze.data(logfile = "datalog.txt")

Error in get.default(where = 1, immediate = T, pas..:
Object "x.3" not found
```

The function expects to find 10 data sets in the working directory and returns an error when it encounters only two. In this example, the log file **datalog.txt** contains the following text:

```
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 may crash after running through many iterations. For example, a rare sequence of random variables may trigger a bug in the function, the function may run out of memory, or the computer may have to be rebooted for unrelated reasons. Because of this possibility, you should write your simulation function so that it can be restarted at a point near where it crashed. This requires you to ensure that the current state of the simulation is saved to disk periodically, and that you can use the recorded information to restart the function. Often, the required state information is simply the iteration number. If you are using random number generators, however, the current seed of the generator `.Random.seed` must be saved as well. The value of `.Random.seed` is updated every time a random number is generated; like any other data set, the updated value is not committed to disk until the successful completion of a top-level expression.

EXAMPLE: A SIMPLE BOOTSTRAP FUNCTION

In this section, we develop a simple bootstrap function to demonstrate some of the ideas for efficient simulations that we discuss in this chapter. The basic purpose of the looping procedures in the `simple.bootstrap` function below is to do blocks of bootstrap samples within a `for` loop. The size of the blocks involves the following trade-off:

- Small block sizes require Spotfire S+ to save `.Random.seed` many times. Because `.Random.seed` is stored on a permanent database, each call to a sampling function uses 81 bytes of memory that are not reclaimed until all functions and loops have finished.
- Large block sizes require Spotfire S+ to store a large matrix of indices.

The speed of `simple.bootstrap` is not very sensitive to the block size except at the extremes. Very small block sizes cause Spotfire S+ to call the `sample` function too often, while very large `n` and `block` values may require more memory than is available in RAM. When this occurs, the computations are forced to use *virtual memory* (or *swap space*). This phenomenon, called *paging*, significantly slows the progress of the computations.

In addition to generating blocks of bootstrap samples, there are a number of subtle points in `simple.bootstrap` that affect memory usage. We note these in the code's comments below. The function is written so that a user interrupt saves as many blocks of bootstrap results as have been completed. It also saves results after exits due to the particular memory problem of allocating too large a data set for the indices.

Note

The default block size in the code below may need to be reduced if `n` is large. For example, if `n*block*6+32` is greater than `options("object.size")`, you should consider changing the default value for the `block` argument.

```
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 is 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({                             # In case function is interrupted
    cat("Saving replications 1:", (i-1)*block, " to .bootstrap.results\n")
    assign(".bootstrap.results", replicates,where = 1, immediate = T)
  })
  for(i in 1:nblocks){                    # Do block samples simultaneously
    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()
  return(result)
}
```

Figure 15.1: *A simple, interruptible bootstrap function.*

SUMMARY OF PROGRAMMING TIPS

Some of the key points from this chapter are:

- When working with large numbers of iterations in a `for` loop, replace the body of the loop with a function that accomplishes the same thing. This delays the slowdown that occurs when the number of iterations becomes larger than approximately 10,000.
- The `lapply` function is currently the most efficient way to do looping in Spotfire S+.
- The `For` function can be used to avoid both the overhead of long-running `for` loops, as well as the memory overhead that results from caching data.
- Be sure to include code in your simulation that writes and appends status information to an external file. You should include enough information so that you can restart the simulation if Spotfire S+ or the computer should crash while it is running.
- It is best to reduce the number of calls to sampling functions like `runif`, because repeatedly changing `.Random.seed` is very inefficient.
- A hybrid of `for` and `lapply` can be more efficient than using either one alone in simulations like bootstrapping, where accomplishing everything in a single `lapply` requires storage of huge matrices of random indices.
- Finally, listen to your hard disk. If it is running a lot, you are probably paging and computations will become very slow. In such cases, it is better to do the simulation in smaller parts using `For`, for example, to prevent memory requirements from growing to the point that paging occurs.

EVALUATION OF EXPRESSIONS 16

INTRODUCTION

To this point, we have for the most part simply assumed that Spotfire S+ 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 Spotfire S+ works, from parsing input to printing values. Together with the information in Chapter 2, Data Management, this chapter provides you with a full account of the machinery that runs the Spotfire S+ environment.

SPOTFIRE S+ SYNTAX AND GRAMMAR

When you are using the Spotfire S+ **Commands** window, your keyboard's standard input is directed immediately to the Spotfire S+ parser, which converts the characters you type into expressions that can be evaluated by the Spotfire S+ evaluator. (If you start in batch mode, the parser's input is provided by the 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 plus sign (+). A semicolon (;) can also be used to terminate an expression, but if the expression is incomplete, Spotfire S+ issues an error message. A complete expression is any typed expression that falls into one of the seven broad classifications shown in Table 16.1.

Table 16.1: *Classifications of expressions.*

Class	Expression
Literals	Literals are the simplest objects known to Spotfire S+. 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 subexpressions. 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 classifications.

Literals

All literals fall into one of the following six categories:

1. Numbers
2. Strings
3. Names
4. Comments
5. Functions
6. Symbolic constants

Numbers

Numbers are further subdivided into *numeric* and *complex* values. Numeric values represent real numbers and can be expressed in any of the following forms:

- As ordinary decimal numbers, such as -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 missing value `NA` (which can be logical or numeric).
- 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.
- As the IEEE special value `NaN` that results from `Inf/Inf`, `0/0`, `Inf-Inf`, and other indeterminate expressions. `NaN` generally prints as `NA`, although you can use `is.nan()` to distinguish it from a missing value.

Note
Numeric data are stored internally in one of three storage modes: `"integer"`, `"single"`, or `"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.

Complex values are similar to numeric values except that 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.

Strings

Strings consist of zero or more characters typed between two apostrophes (`''`) or double quotes (`""`). Table 16.2 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 16.2: *Special characters.*

Character	**Description**
`\t`	Tab
`\b`	Backspace
`\\`	Backslash
`\n`	New line
`\r`	Carriage return (in general, not needed)
`\"`	Double quotes (")
`\'`	Apostrophe (')
`\###`	ASCII character as an octal number (where # is in the range 0 to 7)

Names

Syntactic names are unquoted strings that (1) do not start with a number and (2) consist of alphanumeric characters and periods (.). As described in Chapter 2, 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`, `assign`, etc.).

Note

Attempts to associate objects with the following reserved names will result in an error:

```
if  else  for  while  repeat  next  break  in  function  return
```

The names of the built-in symbolic constants are also reserved:

```
TRUE  T  FALSE  F  NULL  NA  Inf  NaN
```

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.

Functions

A function consists of the word `function`, a parenthesized set of formal arguments (which may be empty), and a complete expression. Function literals, in general, appear only during function assignment, on the right side of the assignment arrow.

Symbolic Constants

Spotfire S+ reserves the following symbolic constants:

```
TRUE  T  FALSE  F  NULL  NA  Inf  NaN
```

Although these constants are names syntactically, Spotfire S+ prevents assignment to them.

Calls

Most complete expressions involve at least one call. Calls are the expressions that do most of the work in Spotfire S+. They fall into three basic categories:

1. *Simple calls* are calls of the form

    ```
    function-name(arglist)
    ```

2. *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* extracts elements from atomic and recursive data and also extracts named components from recursive data. The parser interprets subscript expressions as simple calls, converting expressions such as

   ```
   object[i]  object$i  object[[i]]
   ```

   into the equivalent function calls

   ```
   "["(object, i)  "$"(object, i)  "[["(object, i)
   ```

   Thus, all calls have essentially the same evaluation process.

Spotfire S+ Evaluation

Evaluation of calls is the principal activity of the Spotfire S+ evaluator. Most calls generate at least two frames–the top-level expression frame and the function's evaluation frame. Functions that consist solely of calls to `.Internal`, however, may not generate an evaluation frame. Calls to these functions are called *quick calls* because they avoid the overhead of creating an evaluation frame (which is not needed because there are no Spotfire S+ assignments in the function body and the arguments are matched in the top-level expression frame). However, memory allocated in them may accumulate in the caller's frame.

To evaluate the call, Spotfire S+ 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 Spotfire S+ 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 `sys.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.

As previously mentioned, the value of the function call 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 freed.

Internal Function Calls

Functions defined as calls to `.Internal` are evaluated somewhat differently from calls to functions written wholly in Spotfire S+ or to functions using the interfaces to C, Fortran, or the operating system. 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 a S-PLUS object that is the value of the call.

A few internal functions (for example, `substitute`) do not evaluate their arguments. These functions pass the entire unevaluated expression to the internal C code, which then performs the computations. Most of these functions, which are listed in Table 16.3, use the form of the expression itself to determine how the evaluation is to proceed.

Table 16.3: *Internal functions.*

Function	**Description**
`&&, ||`	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.

Table 16.3: *Internal functions. (Continued)*

Function	Description
`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 with the value of an object with the same name in a specified list or frame, by default, the local frame. Unlike most functions that 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.
`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`, and then evaluates the recalled definition in the new frame.

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 a subscripting operation.

Nested replacements, in which several function calls are nested on the left-hand side, are common. For example, consider the following expression:

```
dimnames(state.x77)[[2]][8] <- "Land Area"
```

This expression extracts the eighth element of the second list element in the `dimnames` attribute of the `state.x77` data set and replaces it with the value on the right-hand side.

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 form

```
"f<-"(x,value)
```

Note

You cannot do

```
x <- "f<-"(x, value=value)
```

for most replacement operations; the above is just a schematic view.

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. Note that the data to be altered must already be in that frame.

Nested replacements are expanded internally into an equivalent series of simple replacements. For example, the expression

```
dimnames(state.x77)[[2]][8] <- "Land Area"
```

might be expanded as follows:

```
assign("..a", dimnames(state.x77)[[2]], frame=Nframe)
..a[8] <- "Land Area"
assign("..b", dimnames(state.x77), frame=Nframe)
..b[[2]] <- ..a
dimnames(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 the 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, Spotfire S+ returns `NULL`.

Loops and Flow of Control

Spotfire S+ 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 flow-of-control instructions.

Because Spotfire S+ has no explicit jumps or `GOTO`s, flow of control is handled by flags that are checked by the evaluator each time it is recursively called. There are three flags—`Break.flag`, `Next.flag`, and `Return.flag`—that correspond to the three flow-of-control instructions `break`, `next`, and `return`.

As we saw on page 498, when a `return` expression is evaluated, `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 `Break.flag` or `Return.flag` is `TRUE`. If `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 `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.

Grouping

Two types of grouping affect Spotfire S+ 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.

THE VALIDATION SUITE 17

INTRODUCTION

You can check the accuracy of Spotfire S+ algorithms and routines as they run on your system by using the `validate` function. The `validate` function draws upon a suite of validation tests that 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 `validate`, using the supplied test files as templates.

This chapter details the coverage of the supplied routines, describes the syntax of `validate`, 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 17.1 describes the coverage of the built-in validation tests. In the left column of the table, tests are grouped into functional areas. The middle column lists the high-level S-PLUS functions tested, and the right column contains brief descriptions of the different cases covered by the tests, when relevant. It is possible to expand the scope of the tests provided, and you can also write routines for other S-PLUS functions (including your own functions).

Table 17.1: *Built-in validation tests.*

Functional Area	**High-Level Functions**	**Test Cases**
ANOVA	`aov`	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.
	`manova`	Simple. Repeated measures.

Descriptive Statistics	`mean`
	`median`
	`var`
	`standard dev.`
		Descriptive statistics for various numeric vectors, including very large and very small values.

	`cor`	Simple correlation coefficient between two vectors.

Table 17.1: *Built-in validation tests.*

Hypothesis Tests	`binom.test`	2-sided alternative. Both 1-sided alternatives.
	`prop.test`	2-sample test, 2-sided alternative.
	`chisq.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 with 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.
	`mantelhaen.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.

Table 17.1: *Built-in validation tests.*

	`var.test`	2-tailed variance ratio test, 2-tailed and 1-tailed (less) alternatives.
	`ks.gof`	1-sample test, 2-sided and both 1-sided alternatives.
	`chisq.gof`	Testing continuous data from a normal distribution with given μ and σ, where expected values are the same in each of the specified number of classes. Testing discrete data from a normal distribution, where expected values are calculated from specified class endpoints.
Multivariate	`princomp`	Components based on both correlations and covariances.
	`factanal`	Principal component extraction with varimax rotation. Maximum likelihood factor solutions with varimax rotation.
Regression	`lm`	Simple linear regression, including hypothesis testing. Multiple regression including hypothesis testing. Polynomial regression. Multiple regression with no intercept, including hypothesis testing.
	`glm`	Logistic regression, including chi-square goodness of fit. Log-linear regression. Gaussian linear model.
	`nls`	Michaelis-Menten model with 4 parameters.

Table 17.1: *Built-in validation tests.*

	`lme`	Random intercept. Random intercept with AR(1) errors. Random factors. Random factors with AR(1) errors.
Statistical Distributions	`pnorm, qnorm`	z-values between -3.0902 and 3.0902.
	`pchisq`	Probabilities from 0.001 to 0.995; degrees of freedom from 1 to 25 by 5, and 30 to 100 by 10.
Survival Analysis	`survfit`	Kaplan-Meier estimator with 2 groups.
	`survdiff`	Log-rank test.
	`coxph`	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.
	`survreg`	Gaussian, Weibull, and exponential models.

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 **/splus/lib/validate** under your Spotfire S+ home directory. 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 17.1: `anova`, `descstat`, `hypotest`, `multivar`, `regress`, `sdistrib`, and `survival`. For example, the following command runs the analysis of variance and regression tests. The argument `verbose` is set to `TRUE`, so the details of each test are returned by `validate`.

```
> validate(file = c("anova", "hypotest"), verbose = T)
```

```
--------------- Analysis of Variance ---------------
{
cat("-------------- Analysis of Variance --------------\n")
T
}
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:
  # in Windows
  Test Directory:C:\splus8\splus\lib\validate
  File splus8\splus\lib\validate\anova: All tests PASSED
  File splus8\splus\lib\validate\hypotest: All tests PASSED
```

To run customized tests, you must first write a test file: for information on creating test files, see the section Creating Your Own Tests (page 516). To use `validate` with your test files, specify the name of your files with the `file` argument and the directory containing them with the `test.loc` argument. For example, suppose you create tests named `anova1` and `hypotest1` in the Windows® directory **C:\Spluswork\valdir**. Your call to `validate` would look like the following:

```
> validate(file = c("anova1", "hypotest1"),
+  test.loc = "C:\\Spluswork\\valdir")
```

Should any of the validation tests 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:
# in Windows:
Test Directory:C:\Spluswork\valdir
File Spluswork\valdir\anova1: All tests PASSED
File Spluswork\valdir\hypotest1: 1 test(s) FAILED
```

CREATING YOUR OWN TESTS

The built-in validation tests are set up as *loop tests*. A loop test consists of a set of Spotfire S+ commands that are grouped as a braced expression; the test returns either TRUE or FALSE depending on the outcome. A validation test file contains one or more of these loop-style expressions. For example, the code for the first loop test in the built-in validation test for descriptive statistics is below. The code tests the S-PLUS function mean:

```
{
# Function: mean
# Data: test.mat; a test data set suggested by Leland
# Wilkinson in Statistic's Statistics Quiz (1985).
# Reference(s): Sawitzki, G. 1993. Numerical Reliability of
# Data Analysis Systems. submitted for publication in
# Computational Statistics and Data Analysis.
# Description: test mean for numeric data
tol <- 1e-6
test.mat <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 0, 0, 0,
0, 0, 0, 0, 0,
99999991, 99999992, 99999993, 99999994, 99999995, 99999996,
99999997, 99999998, 99999999,
0.99999991, 0.99999992, 0.99999993, 0.99999994, 0.99999995,
0.99999996, 0.99999997, 0.99999998, 0.99999999,
1e+12, 2e+12, 3e+12, 4e+12, 5e+12, 6e+12, 7e+12, 8e+12,
9e+12, 1e-12, 2e-12, 3e-12, 4e-12, 5e-12, 6e-12, 7e-12,
8e-12, 9e-12,
0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5),
ncol=7, dimnames = list(NULL, c(
  "X", "Zero", "Big", "Little", "Huge", "Tiny", "Round")))
test.mean <- matrix(0,1,7)
test.mean[1] <- mean(test.mat[,1])
test.mean[2] <- mean(test.mat[,2])
test.mean[3] <- mean(test.mat[,3])
test.mean[4] <- mean(test.mat[,4])
test.mean[5] <- mean(test.mat[,5])
test.mean[6] <- mean(test.mat[,6])
test.mean[7] <- mean(test.mat[,7])
all(c(test.mean[1] == 5,
      test.mean[2] == 0,
      test.mean[3] == 99999995,
      test.mean[4] == 0.99999995,
      test.mean[5] == 5.0e+12,
      test.mean[6] == 5.0e-12,
      test.mean[7] == 4.5))
}
```

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 Spotfire S+ can be found in **SHOME/splus/lib/validate**; the code above is the first braced expression of the file **descstat** in this directory. If the variable **SHOME** is not set in your environment, use the path returned by calling `getenv("SHOME")` within Spotfire S+.

The basic construct of a test file includes an expression, which is defined by a set of commands enclosed in curly brackets `{}`. The `validate` function expects the first expression to be a comment; if the first expression is a test, it is not 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 `TRUE` or `FALSE`.

INDEX