



TIBCO Spotfire S+[®] 8.2 Guide to Graphics

November 2010

TIBCO Software Inc.

IMPORTANT INFORMATION

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN *TIBCO SPOTFIRE S+® LICENSES*). USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO Software Inc., TIBCO, Spotfire, TIBCO Spotfire S+, Insightful, the Insightful logo, the tagline "the Knowledge to Act," Insightful Miner, S+, S-PLUS, TIBCO Spotfire Axum, S+ArrayAnalyzer, S+EnvironmentalStats, S+FinMetrics, S+NuoOpt, S+SeqTrial, S+SpatialStats, S+Wavelets, S-PLUS Graphlets, Graphlet, Spotfire S+ FlexBayes, Spotfire S+ Resample, TIBCO Spotfire Miner, TIBCO Spotfire S+ Server, TIBCO Spotfire Statistics Services, and TIBCO Spotfire Clinical Graphics are either registered trademarks or trademarks of TIBCO Software Inc. and/or subsidiaries of TIBCO Software Inc. in the United States and/or other countries. All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. This

software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. Please see the readme.txt file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

Copyright © 1996-2010 TIBCO Software Inc. ALL RIGHTS RESERVED. THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

TIBCO Software Inc. Confidential Information

Reference

The correct bibliographic reference for this document is as follows:

TIBCO Spotfire S+® 8.2 Guide to Graphics 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.	<i>Installation and Administration Guide</i>
Want to review the third-party products included in Spotfire S+, along with their legal notices and licenses.	<i>Licenses</i>

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 [®] .	<i>Getting Started Guide</i>
Are a new Spotfire S+ user and need how to use Spotfire S+, primarily through the GUI.	<i>User's Guide</i>
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).	<i>Spotfire S+ Workbench User's Guide</i>
Have used the S language and Spotfire S+, and you want to know how to write, debug, and program functions from the Commands window.	<i>Programmer's Guide</i>
Are familiar with the S language and Spotfire S+, and you want to extend its functionality in your own application or within Spotfire S+.	<i>Application Developer's Guide</i>
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.	<i>Guide to Graphics</i>
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.	<i>Big Data User's Guide</i>
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.	<i>Guide to Packages</i>

Spotfire S+ documentation. (Continued)

Information you need if you...	See the...
Are looking for categorized information about individual S-PLUS functions.	<i>Function Guide</i>
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.	<i>Guide to Statistics, Vol. 1</i>
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+.	<i>Guide to Statistics, Vol. 2</i>

CONTENTS

Important Information	ii
TIBCO Spotfire S+ Books	iv
	vii
Chapter 1 Graphics Enhancements	1
Overview	2
Color Specification	3
Additional Graphics Arguments	17
Vectorized Graphics Parameters	21
Backward Compatibility	24
Chapter 2 Traditional Graphics	27
Introduction	29
Getting Started with Simple Plots	30
Frequently Used Plotting Options	33
Visualizing One-Dimensional Data	43
Visualizing the Distribution of Data	49
Visualizing Three-Dimensional Data	57
Visualizing Multidimensional Data	62
Interactively Adding Information to Your Plot	66
Customizing Your Graphics	72
Controlling Graphics Regions	79

Controlling Text and Symbols	83
Controlling Axes	89
Controlling Multiple Plots	95
Adding Special Symbols to Plots	102
Traditional Graphics Summary	108
References	113
Chapter 3 Traditional Trellis Graphics	115
A Roadmap of Trellis Graphics	117
Giving Data to Trellis Functions	120
General Display Functions	124
Arranging Several Graphs on One Page	146
Multipanel Conditioning	148
General Options for Multipanel Displays	162
Scales and Labels	166
Panel Functions	171
Panel Functions and the Trellis Settings	175
Superposing Multiple Value Groups on a Panel	179
Aspect Ratio	189
Data Structures	194
Summary of Trellis Functions and Arguments	198
Chapter 4 Editing Graphics in UNIX	205
Introduction	206
Using motif Graphics Windows	207
Using java.graph Windows	221
Printing Your Graphics	233
Chapter 5 Editing Graphics in Windows	249
Graphs	250

Formatting a Graph	264
Working With Graph Objects	286
Plot Types	288
Using Graph Styles and Customizing Colors	291
Embedding and Extracting Data in Graph Sheets	293
Linking and Embedding Objects	294
Printing a Graph	297
Exporting a Graph to a File	298
Chapter 6 Windows Editable Graphics Commands	301
Introduction	303
Getting Started	305
Graphics Objects	308
Graphics Commands	313
Plot Types	322
Titles and Annotations	345
Formatting Axes	350
Formatting Text	352
Layouts for Multiple Plots	357
Specialized Graphs Using Your Own Computations	359
Chapter 7 Working With Graphics Devices	361
Introduction	362
The graphsheet Device	363
The motif Device	364
The java.graph Device	365
The pdf.graph Device	374
The wmf.graph and emf.graph Devices	383
The postscript Device	390

Device-Specific Color Specification	397
Chapter 8 Spotfire S+ Graphlets[®]	413
Introduction	414
Creating a Graphlet Data File	418
Embedding the Graphlet in a Web Page	430
Using the Graphlet	436
Index	441

GRAPHICS ENHANCEMENTS

1

Overview	2
R Graphics Compatibility	2
Color Specification	3
Color Names	5
Global Color Palette	9
Other Color Spaces	14
Summary of Color Specification Functions	15
Additional Graphics Arguments	17
Additional Color Arguments	17
Additional Text-Related Arguments	19
Vectorized Graphics Parameters	21
Backward Compatibility	24
Using Legacy Graphics	24
Using Device-Specific Palettes	24

OVERVIEW

S-PLUS 8.0 introduced a variety of graphics enhancements related to:

- Color specification
- Additional color arguments for plot elements such as titles, axes, and points
- Additional vectorized parameters

The primary emphasis was on enhanced color handling, with some additions to other graphics arguments.

Note
The enhancements described in this chapter apply <i>only</i> to Spotfire S+ command line graphics; they do not apply to graphics produced in the Windows graphical user interface.

R Graphics Compatibility

The graphics enhancements are largely inspired by features of the R system for statistical computation and graphics. In general, the changes introduced here are intended to improve compatibility between S-PLUS and R.

Areas where S-PLUS and R differ are addressed in this chapter.

COLOR SPECIFICATION

Devices and Palettes

When S and S-PLUS were initially developed, color support varied among graphics devices. For example, the number of available colors and the way colors were specified differed between a Hewlett-Packard HP-GL plotter and a Hewlett-Packard 2623 graphics terminal.

The HP pen plotter supported 8 colors, corresponding to the pens in stables 1 through 8. The software using the plotter had no knowledge of the actual color of each pen. S-PLUS would specify the number of the pen to use.

As UNIX matured, the motif graphics device supported a larger number of colors. However, the system as a whole was often limited to a total of 256 colors. These were shared among S-PLUS and other applications such as Netscape. S-PLUS specified a color that was mapped to an available system color through a resource file.

The mapping of a color index to an actual value was done through a *color palette*, which is simply a lookup table that determines the color to use for each index value.

In early versions of Windows, 16 colors were typically available. Higher color resolution often required more RAM, video RAM, or processing time than was available. As of Windows 95, the recommendation was VGA or higher, with 256-color SVGA recommended. A palette of 256 colors could be defined at the operating-system level, and these were the only colors available.

Because of these platform and device specific aspects of color specification, the S-PLUS graphics system left the mapping of color indexes to color values to the device. S-PLUS graphics commands referred to colors only as color 0, 1, 2, 3, etc., and the mapping of these indexes to actual color values was performed by the device.

Under this approach, different color specification systems evolved for the various devices.

Widespread Truecolor Availability

As computing power has increased, most displays and file formats can support at least 24-bit *Truecolor*, in which a color is specified as a triplet of red, green, and blue (RGB) values, each between 0 and 255.

Because this wide variety of colors is readily available, it is no longer necessary for Spotfire S+ to refer to a color as an index that gets mapped to a value by the device. Spotfire S+ can refer to the color by its RGB value.

Colors may now be specified using either a numeric index into a palette as before, or through a string specifying a color name.

RGB Values

By convention, an RGB triplet is represented by a string in the form "#FF0000", where "#" indicates that the subsequent character pairs are hexadecimal encodings of the 256 potential values for red, green, and blue respectively.

For example, "#483D8B" represents dark slate blue. This corresponds to the RGB triplet {72, 61, 139}.

- The red value is obtained as: 48 (base 16) = 72 (base 10).
- The green value is obtained as: 3D (base 16) = 61 (base 10).
- The blue value is obtained as: 8B (base 16) = 139 (base 10).

The term *RGB value* refers to a string RGB specification in this format.

Transparency

Some devices can support 32-bit color, in which an *alpha channel* value is added to specify transparency. On a scale of 0 to 255, an alpha of 0 is fully transparent while 255 is fully opaque. Other alpha values specify a translucent color, sometimes referred to as semi-transparent or semi-opaque.

When a translucent color is drawn, items beneath it are not completely obscured. For example, a Venn diagram can be created by overlapping three translucent circles. The colors of the overlapping areas are a composite of the translucent colors, rather than the color of the topmost circle.

Transparency support refers to the ability to specify transparency and translucency via an alpha channel.

RGBA Values

When transparency is specified, the hex string describing the color has two additional characters that represent the alpha value. So, a semi-transparent red can be specified as "#FF000080". The alpha value is typically not specified when the color is fully opaque (i.e., has an alpha of "FF").

A color description in this format is called an *RGBA value*.

In the discussion that follows, RGB refers to either the three-element RGB specification or the four-element RGBA specification. When specified with the three-element RGB specification the color is fully opaque.

Device Support for Transparency

The `java.graph` and `pdf.graph` devices support alpha channel transparency. When creating files, the `java.graph` device provides alpha channel support for PNG and TIFF files only. The JPEG, PNM, and BMP formats do not support transparency.

The `motif`, `wmf.graph`, `emf.graph`, and `graphsheet` devices do not support transparency as transparency is not available in the windowing libraries they use (Xlib and Windows GDI respectively). The `postscript` device does not support transparency as it is not available in the Postscript format.

Devices that do not support transparency do not draw items that are fully transparent (i.e., have an alpha of "00"), and treat all other alpha values as fully opaque.

Color Names

Colors can be represented as strings in the following formats:

- A color name, such as "red" or "darkslateblue"
- An RGB value, such as "#FF0000" for red or "#483D8B" for dark slate blue

The CSS3 Color Module specification is used to convert color names to RGB values. This is the color naming system used in HTML 4.01 and SVG 1.0. It provides a set of 147 named colors, and is described at:

<http://www.w3.org/TR/css3-color/>

These are referred to as the *CSS color* names.

Using Named Colors

Here is a plot that specifies a color by name:

```
> plot(corn.rain, corn.yield, col="maroon")
```

And here is the same plot using a different color specified by a hex RGB value:

```
> plot(corn.rain, corn.yield, col="#00FA9A")
```

Color String Format

Color names must start with a letter. All operations with color names are case-insensitive, so "red", "Red", and "RED" are treated as identical names. When storing and returning color names, lowercase is used and spaces are removed, so "Dark Slate Blue" is converted to "darkslateblue".

The RGB values "#483D8B" and "#483d8b" are both valid formats for specifying a color. RGB values are returned in uppercase (e.g., "#483D8B").

Getting Color Information

The `colors` function returns the names of the colors. The `color.values` function returns the RGB values, with the corresponding color names as the vector element names.

To get the color names:

```
> colors()
```

To get the color RGB values for all named colors:

```
> color.values()
```

To get the RGB values for specific colors:

```
> color.values(c("red", "teal"))
```

The RGB value is a seven-character string in the form "#FF0000" if the color is fully opaque (i.e., if the alpha is "FF" or unspecified). If the color is semi-transparent, it is specified as a nine-character string in the form "#FF000080". Because transparency is not supported by all graphics devices, seven-character representation is more common. The named colors do not include alpha specification in their mapped values.

CSS and X11 Colors

An alternative mapping of color names to values is the X11 color specification, as defined in the `<X11root>/lib/rgb.txt` file in an X Window System installation.

X11 color mapping includes 752 color names, of which there are 95 pairs with and without embedded spaces that map to the same value. For example, the second and third colors listed are "ghost white" and "GhostWhite". R uses the X11 color specification with names normalized to lowercase with no embedded spaces. This yields 657 colors.

The CSS and X11 specifications differ in their RGB values for four colors: "Gray", "Green", "Maroon", and "Purple". The CSS versions are darker. Table 1.1 specifies the differences.

Table 1.1: *CSS and X11 RGB Value Conflicts*

Color Name	CSS RGB Value	X11 RGB Value
"Gray"	"#808080"	"#BEBEBE"
"Green"	"#008000"	"#00FF00"
"Maroon"	"#800000"	"#B03060"
"Purple"	"#800080"	"#A020F0"

The 657 X11 colors include gradations for several of the named colors. For example, the five shades of maroon are: "maroon", "maroon1", "maroon2", "maroon3", and "maroon4". Shades of gray range from "gray0" through "gray100". Because color shades can be specified using their hex representation, the CSS specification does not include this type of gradation naming.

In addition to the various shades, X11 includes the colors "lightgoldenrod" and "violetred", which are not in the CSS table. It also includes "navyblue" as another name for "navy".

The CSS table includes eight color names not available in the X11 table: "aqua", "crimson", "fuchsia", "indigo", "lime", "olive", "silver", and "teal".

Setting the Color Names

By default Spotfire S+ uses the CSS color names, however, color name mapping can be customized. For example, you can specify a different set of color names to be used, such as the X11 color names. You can also add your own custom color names, such as the corporate color "upsbrown" or "rojo", the Spanish name for red.

The `add.color.values` function modifies the color name mapping. It takes a named character vector of RGB values in the form `"#FF0000"` or `"#FF000080"`. The names are used as the color names, and the values as the RGB values.

Additional arguments determine whether to discard the current table of names, and whether the old or new color definition should be used when a specified name is already in the table.

To add `"upsbrown"` and `"rojo"` as new colors:

```
> add.color.values(c(upsbrown="#964B00", rojo="#FF0000"))
```

Preconstructed Color Sets

Spotfire S+ includes preconstructed color sets that can be used with the `add.color.values` function to access common color mappings:

- `css.colors`: named character vector of the 147 CSS colors
- `x11.colors`: named vector of the 657 X11 colors, with names normalized to lowercase and no embedded spaces

In the following examples, the assumption is that the default CSS colors are in place before each expression is executed. The examples are not cumulative.

To use the X11 color definitions to resolve names such as `"gray87"` that are not included in the CSS color set:

```
> add.color.values(x11.colors)
```

The above command generates warnings to indicate which colors in the new set will be ignored because there are already colors of the specified names with different values. To prevent these warnings, include the `warn=F` argument:

```
> add.color.values(x11.colors, warn=F)
```

To give priority to the X11 definitions where they differ from CSS, include the `overwrite=T` argument:

```
> add.color.values(x11.colors, overwrite=T)
```

To discard the CSS colors altogether and use only the X11 definitions, include the `new=T` argument:

```
> add.color.values(x11.colors, new=T)
```

Color Name Resolution

When a color is specified as a string, the following rules are used to convert it to an RGB value:

1. If the string begins with "#" and includes six subsequent characters, it is interpreted as a hex representation of an RGB value specifying a fully opaque color (equivalent to specifying an alpha value of "FF").
2. If the string begins with "#" and includes eight subsequent characters, it is interpreted as a hex representation of an RGBA value.
3. If the string begins with a letter, the color names map is searched for the name, and the RGB value for the first match is used.
4. If the string is "transparent", "NA", or the empty string "" it is interpreted as representing a fully transparent point. The fully transparent pure white color "#FFFFFF00" is used.
5. If the string can be converted to an integer, such as 2, it is converted to an integer and used to look up a color in the global color palette described in the next section.

If no match is found, a warning is generated and the fully opaque pure black color "#000000FF" is used. The warning message specifies the name that could not be found.

Global Color Palette

As discussed above, prior to Spotfire S+ 8, the mapping of color indexes to specific colors was left to individual devices. Due to differences in the ways various devices specified colors, it could be complex to map indexes to identical colors in all devices.

With the addition of RGB color specification to all devices, it is now possible to use a *global color palette* rather than device-specific palettes.

The global color palette is simply a vector of RGB values. Whenever a color is specified by a numeric index, Spotfire S+ uses the global color palette as the lookup table for mapping indexes to specific colors. For example:

```
> plot(corn.rain, corn.yield, col=2)
```

The palette is set and retrieved using the `palette` function. To get the character vector giving the current palette, call this function with no arguments:

```
> palette()
[1] "#EDF8E9" "#C7E9C0" "#A1D99B" "#74C476" "#41AB5D"
[6] "#238B45" "#005A32"
```

To set the palette, specify a character string of palette values:

```
> palette(mypalette)
```

In this case, the vector element names are ignored and the values are used to set the color mapping between integers 1, 2, 3, etc., and RGB values. The palette vector can be of any length. If the index is greater than the number of colors in the palette, the palette colors are replicated to the necessary length.

Color values are usually specified as RGB values, but may be specified as names or indexes. In all cases, the value is immediately converted to the RGB value and stored in that form. This rule also applies for the values provided to the `add.color.values` function.

Image Color Palette

A separate *image color palette* is used for situations in which a larger number of colors is required to indicate a gradation of values. These are used for image plot, hexagonal binning, level plots, and draped wireframes.

The `image.palette` function sets and retrieves the image color palette. It is used in the same manner as the `palette` function.

To get the image palette, use the function with no arguments:

```
> image.palette()
```

To set the image palette, provide a character vector specifying colors in hex RGB format:

```
> image.palette(topo.colors(10))
```

For information about `topo.colors` see the section *Creating Color Sets* on page 13.

The number of image colors may vary. Unlike the standard palette, image colors are not replicated to match a set of indexes. Instead, the values to plot are grouped by intervals and all values in an interval are assigned one of the image colors.

Conceptually, the range of values is mapped to the interval 0 to 1. This is divided by the number of image colors, and each segment of the range is assigned a color. For example, if there are 5 image colors, data that are mapped to values between 0.2 and 0.4 are displayed in the second image color.

R does not have an image color palette. In R, functions such as `image` have a `col` argument with a hardcoded default set of colors, and a different set of colors is set by explicitly providing a different set of values in the call to `image`.

For compatibility with R, a `col` argument has been added to the S-PLUS `image` function.

Trellis functions such as `contourplot`, `levelplot`, and `wireframe` use the `trellis.par.get("regions")` command to get image colors. When the global image palette is used, the `trellis.par.get` function will use this palette to get the region colors, rather than the `trellis.settings` object.

Background Color

The background color for the current graphics device can be set with the `bg` parameter:

```
> par(bg="red")
```

The background color is used when a color index of 0 is specified. For example, if you want to set every third point to match the background color, you could plot a graph as follows:

```
> plot(1:10, type="n", bg="white")
> points(1:10, col=c(1,2,0))
```

Default Palettes

The default palette colors evolved from the Trellis color scheme initially defined by Bill Cleveland for color printing. The darkness of these colors was adjusted for better display on a computer screen using a white background.

The default image colors are a range of 256 navy blue values selected to provide good contrast between low and high values, while allowing perception of differences between midrange values.

Spotfire S+ includes vectors corresponding to these sets of colors:

- `splus.default.colors` is a named vector of 16 palette colors: "sblack", "sdarkblue", "sdarkred", "sgreen", "sorange", "sblue", "sbrown", "sbrick", "slightcyan", "slightmagenta", "slightgreen", "slightorange", "slightblue", "slightyellow", "slightcoral", "sgray".
- `splus.default.image.colors` is an unnamed vector of 256 shades of navy used as the default image colors.

To reset the palette and image palette to their defaults:

```
> palette(splus.default.colors)
> image.palette(splus.default.image.colors)
```

A shorthand method for restoring the defaults is to specify the value "default" with these functions:

```
> palette("default")
> image.palette("default")
```

This special treatment of the value "default" is also present in R.

The `splus.default.colors` vector has names to provide a consistent way to refer to the default colors, however, they are not automatically set as known color names. To use the default color names, first add them to the set of known named colors:

```
> add.color.values(splus.default.colors)
```

Palettes Matching R

The default R palette colors are the following eight colors from the X11 color scheme: "black", "red", "green3", "blue", "cyan", "magenta", "yellow", and "gray".

The default R image colors are generated with `heat.colors(12)`. For information about `heat.colors` see the section *Creating Color Sets* on page 13.

The following commands set the palette and image palette to match the palettes in R:

```
> rcolors <- x11.colors[c("black", "red", "green3",
+ "blue", "cyan", "magenta", "yellow", "gray")]
> palette(rcolors)
> image.palette(heat.colors(12))
```

To use the X11 names first when resolving names, with the CSS names only used for names not in the X11 names:

```
> add.color.values(x11.colors, replace=T)
```

You can use the following to set the Spotfire S+ palette and image palette to use the R default colors:

- `r.default.colors`: a named character vector that contains the RGB hex string values for the 8 colors in the default R global color palette.
- `r.default.image.colors`: a named character vector that contains the RGB hex string values for the 12 colors R uses by default in its image function.

To set the palette and image palette to use the R defaults:

```
> palette(r.default.colors)
> image.palette(r.default.image.colors)
```

Creating Color Sets

The following functions are useful for generating sets of image colors:

- `rainbow`
- `heat.colors`
- `terrain.colors`
- `topo.colors`
- `cm.colors`
- `gray`
- `grey`
- `gray.colors`
- `grey.colors`

These functions are also available in R. For compatibility, the S-PLUS functions return the same values as their R counterparts.

The `rainbow` function creates colors from across the spectrum. By default, the hue is varied while the saturation and value remain constant. These colors are well suited for points, lines, and bars. They can be used to create an image palette, but that is not their primary purpose.

The `cm.colors`, `heat.colors`, `terrain.colors`, and `topo.colors` functions generate color sets typically used for image colors:

- `cm.colors`: cyan to magenta with a middle value of white
- `heat.colors`: red to white through orange and yellow
- `terrain.colors`: green to white through peach earth-tones
- `topo.colors`: the first third of the range is medium to light blue; the middle third is medium-green to yellow-green; the last third is pure yellow to navajo white

The `gray`, `grey`, `gray.colors`, and `grey.colors` functions create a sequence of gray values in which the red, green, and blue values are all equal.

Other Color Spaces

Computer monitors typically display color by mixing various intensities of red, green, and blue pixels. Therefore, computer colors are typically described in the red, green, and blue (RGB) color space. However, color spaces can be parameterized in other ways.

HSV

The hue, saturation, and value (HSV) color space defines a color in terms of the following properties:

- Hue is the color type, such as red, blue or yellow. Varying hue from 0 to 100% creates a full color wheel.
- Saturation is the vibrancy of the color, referred to as the *purity*. The lower the saturation, the more grayness is present and the more faded the color appears.
- Value is the brightness of the color.

HSV is a nonlinear transformation of the RGB color space that is useful for creating sequences of values. For example, if you want to create a sequence of various shades of the same color, you can specify a range of values with the same hue and vary the saturation and/or brightness.

HSV is also known as the hue, saturation, and brightness (HSB) color space.

The `hsv` function creates a set of RGB colors based on a hue, saturation, and value description. The `rgb2hsv` function converts RGB values to the corresponding HSV triplets. These functions are also available in R.

HSL

The hue, saturation, and lightness (HSL) color space defines the ranges for saturation and brightness differently:

- In HSL, saturation runs from fully saturated to the equivalent gray. In HSV, saturation runs from saturated color to white.
- In HSL, lightness runs from black through the chosen hue to white. In HSV, the value only runs from black to the chosen hue.

The `hsl` function creates a set of RGB colors based on a hue, saturation, and lightness description. The `rgb2hsl` function converts RGB values to the corresponding HSL triplets. These functions are not available in R.

Summary of Color Specification Functions

The following functions are available for creating, setting, and retrieving the color specifications.

Set and retrieve color maps and palettes:

- `colors` returns a character vector of the known color names.
- `color.values` gets the RGB or RGBA values for the named colors. (Not available in R.)
- `add.color.values` modifies the table of named colors. (Not available in R.)
- `palette` sets or gets the character vector used to map integer color indexes to string values.
- `image.palette` sets or gets the character vector used to map image index values to string values. (Not available in R.)

Create and convert RGB color descriptions:

- `rgb` converts vectors of numeric red, green, blue, and optionally alpha values to RGB values in the form "#FF0000".
- `col2rgb` converts strings in the form "red" or "#FF0000" to a matrix of numeric red, green, blue, and optionally alpha values.
- `hsv` converts HSV to RGB.
- `rgb2hsv` converts RGB to HSV.
- `hsl` converts HSL to RGB. (Not available in R.)
- `rgb2hsl` converts RGB to HSL. (Not available in R.)

Create a string for use with `palette` or `image.palette`:

- `gray`
- `grey`
- `gray.colors`
- `grey.colors`
- `rainbow`
- `cm.colors`
- `heat.colors`
- `terrain.colors`
- `topo.colors`

For compatibility with R, each of these functions has the same interface and behavior as its R counterpart if a corresponding R function exists.

For backward compatibility of the Spotfire S+ graphics system, two functions are available:

- `use.legacy.graphics` specifies whether to use the graphics enhancements introduced in Spotfire S+ 8 (i.e., the features described in this chapter), or revert to the graphics functionality of S-PLUS 7. For usage details, see the section *Using Legacy Graphics* (page 24).
- `use.device.palette` specifies whether integers for color values are converted to RGB values globally (i.e., the same RGB values are used for all devices) or on a device-specific basis, as in previous versions of Spotfire S+. For usage details, see the section *Using Device-Specific Palettes* (page 24).

ADDITIONAL GRAPHICS ARGUMENTS

This section describes additional graphics arguments that enable you to specify characteristics for plot elements on a more granular basis than was possible in previous Spotfire S+ releases.

Note

Using the `par` function, you can set defaults for all of the arguments described in this section. See the section *Setting and Viewing Graphics Parameters* (page 74) and the `par` online help for more information.

Additional Color Arguments

In previous Spotfire S+ releases, the `col` argument to the `plot` function specified the color for all elements of the plot. This included not only the points, but also the axes, labels, ticks, title, and subtitle. This meant that in order to have red points while using black for the axes and other elements, it was first necessary to use `plot` with `type="n"` to set up the axis, then call the `points` function to draw the points.

Separate color arguments are now available for various plot elements:

- `col`: color of the plot content, such as points and lines.
- `col.axis`: color of axis tick labels.
- `col.lab`: color of x and y labels.
- `col.main`: color of the main title.
- `col.sub`: color of the subtitle.
- `fg`: foreground color for plot elements such as axes, axis tick marks, and boxes.
- `bg`: background color to be used next time a page is created.

Table 1.2 lists functions that honor these arguments, along with an indication of which arguments apply to each function. Note that the functions listed in this table are not exclusive in that additional functions may also recognize these arguments.

Table 1.2: Color Argument Availability by Function

Function	col.axis	col.lab	col.main	col.sub	fg	bg
axes	Yes	Yes	Yes	Yes	Yes	No
axis	Yes	No	No	No	Yes	No
barplot	Yes	Yes	Yes	Yes	Yes	Yes
boxplot	Yes	Yes	Yes	Yes	Yes	Yes
contour	Yes	Yes	Yes	Yes	Yes	Yes
dotchart	Yes	Yes	Yes	Yes	Yes	Yes
hist	Yes	Yes	Yes	Yes	Yes	Yes
image	Yes	Yes	Yes	Yes	Yes	Yes
pairs	Yes	Yes	No	No	Yes	Yes
persp	Yes	Yes	Yes	Yes	Yes	Yes
pie	No	Yes	Yes	Yes	No	Yes
plot	Yes	Yes	Yes	Yes	Yes	Yes
qqnorm	Yes	Yes	Yes	Yes	Yes	Yes
qqplot	Yes	Yes	Yes	Yes	Yes	Yes
title	No	No	Yes	Yes	No	No

Yes = supported

No = not supported

R Compatibility

The new color parameters are also available in R, but their use as of R 2.2.0 is inconsistent. For example, the `boxplot` function honors the `col.axis` parameter but ignores the `fg` parameter.

Some R functions use these parameters in different ways within plotting functions than within the `par` function. For example, `plot` has a `bg` parameter indicating the background color for filled symbols.

In Spotfire S+, the intent is to use these parameters consistently. Where R is inconsistent, the behavior of R and S-PLUS differs.

Trellis Functions

The Trellis functions such as `xyplot` have always allowed specification of colors for plot elements by providing a list with a `col` element as the `xlab`, `ylab`, `main`, `sub`, and/or `scales` argument. For example:

```
xyplot(Mileage~Weight, fuel.frame,  
       main=list("My title", col=3))
```

In addition, colors may also be set in the relevant `trellis.settings` object.

Because Trellis already has a system for setting different colors for various elements of the plot, the new color parameters are not used by the Trellis functions.

Note that the Trellis functions support specifying color by name. For an example, see the section Trellis Functions on page 22.

Additional Text-Related Arguments

Separate arguments are available for controlling various additional characteristics of the text elements of a plot:

- `cex.axis`: character expansion (size) for axis tick labels
- `cex.lab`: character expansion for x and y labels
- `cex.main`: character expansion for the main title
- `cex.sub`: character expansion for the subtitle
- `font.axis`: font for axis tick labels
- `font.lab`: font for x and y labels
- `font.main`: font for the main title
- `font.sub`: font for the subtitle

Table 1.3 lists the additional text-related arguments and identifies the functions to which they apply.

Table 1.3: *Additional text-related Argument Availability by Function*

Function	cex.axis font.axis	cex.lab font.lab	cex.main font.main	cex.sub font.sub
axes	Yes	Yes	Yes	Yes
axis	Yes	No	No	No
barplot	Yes	Yes	Yes	Yes
boxplot	Yes	Yes	Yes	Yes
contour	Yes	Yes	Yes	Yes
dotchart	Yes	Yes	Yes	Yes
hist	Yes	Yes	Yes	Yes
image	Yes	Yes	Yes	Yes
pairs	Yes	Yes	Yes	Yes
persp	Yes	Yes	Yes	Yes
pie	No	Yes	Yes	Yes
plot	Yes	Yes	Yes	Yes
qqnorm	Yes	Yes	Yes	Yes
qqplot	Yes	Yes	Yes	Yes
title	No	No	Yes	Yes

VECTORIZED GRAPHICS PARAMETERS

In versions prior to Spotfire S+ 8, graphics arguments to `plot` such as `col`, `pch`, and `cex` could only take a single value for all points in the plot. To get different colors for groups of points, it was necessary to first use `plot` with `type="n"` to set up the axis, and then to use `points` within a loop to add points of different colors.

The `plot` function and functions adding individual points to a plot have been enhanced to accept a vector of values for the `col`, `pch`, and `cex` parameters.

Functions that add arrows and line segments to plots now accept a vector of values for the `col`, `lty`, and `lwd` parameters.

Table 1.4 lists functions that honor the vectorized parameters along with an indication of which parameters apply to each function.

Table 1.4: *Vectorized Graphic Parameter Availability by Function*

Function	col	cex	pch	lty	lwd
arrows	Yes	No	No	Yes	Yes
axes	No	Yes	No	Yes	Yes
axis	Yes	Yes	No	Yes	Yes
barplot	Yes	No	No	Yes	Yes
boxplot	Yes	No	No	Yes	Yes
contour	Yes	Yes	No	Yes	Yes
dotchart	Yes	Yes	Yes	No	No
hist	Yes	Yes	No	Yes	Yes
legend	Yes	Yes	Yes	Yes	Yes
lines	Yes	No	No	Yes	Yes
mtext	Yes	Yes	No	No	No

Table 1.4: *Vectorized Graphic Parameter Availability by Function (Continued)*

Function	col	cex	pch	lty	lwd
pairs	Yes	Yes	Yes	No	No
par	No	No	No	No	No
pie	Yes	No	No	Yes	Yes
plot	Yes	Yes	Yes	Yes	Yes
points	Yes	Yes	Yes	No	No
polygon	Yes	No	No	Yes	Yes
qqnorm	Yes	Yes	Yes	No	No
qqplot	Yes	Yes	Yes	No	No
segments	Yes	No	No	Yes	Yes
symbols	No	No	No	Yes	Yes
text	Yes	Yes	No	No	No
title	Yes	Yes	Yes	No	No

If the length of the vector is less than the number of points, the values are replicated to the desired length. The length of the vector does not have to evenly divide the number of values.

Note that the `polygon` function already supported vectors of color values for its `border` and `col` arguments, and each element in these vectors corresponds to a polygon. The same rule is used for the `lty` and `lwd` arguments to the `polygon` function.

Trellis Functions

Trellis functions such as `xyplot`, `qqmath`, `qq`, `splom`, and `stripplot` use a panel function that calls `points` to place the points on the page. The `col`, `pch`, and `cex` arguments are passed directly on to the panel function and then on to `points`.

Support for vectorized parameters in `points` provides this same support to these Trellis functions, so no changes are needed in Trellis to support vectorization. If there is no grouping variable, this will create a correct plot:

```
> d<-data.frame(x=1:60, y=1:60,
+   group=rep(c("A","B","C"),each=20))
> xyplot(y~x, data=d,
+   col=rep(c("red","green","blue"),each=20),
+   pch=rep(0:2,each=20),
+   cex=rep(1:3,each=20))
```

These parameters are passed to the panel function unchanged. When there are multiple panels, the `subscripts` argument must be used to extract the elements of the parameter vectors corresponding to the `x` and `y` elements in a specific panel:

```
> xyplot(y~x|group, data=d,
+   col=rep(c("red","green","blue"),each=20),
+   pch=rep(0:2,each=20),
+   cex=rep(1:3,each=20),
+   panel = function(x, y, subscripts, col, pch, cex) {
+     panel.xyplot(x, y,
+       col=col[subscripts], pch=pch[subscripts],
+       cex=cex[subscripts])
+   }
+ )
```

If the parameters are vectors but are not the same length as `x` and `y`, they should be replicated to the same length before calling the Trellis function in order for the elements to be matched correctly during the subscripting.

BACKWARD COMPATIBILITY

By default, Spotfire S+ 8 uses the graphics functionality described in this chapter. The functions described below allow you to enable backward compatibility with the S-PLUS 7 graphics system.

Using Legacy Graphics

To disable all enhanced graphics functionality (including the global palettes) and revert to S-PLUS 7 graphics functionality, set the `use.legacy.graphics` function to `TRUE` as follows:

```
> use.legacy.graphics(T)
```

Note

Attempting to change the `use.legacy.graphics` setting while a device is open causes an error. First call `graphics.off()` to close all graphics devices, and then call `use.legacy.graphics` to change the setting.

In legacy graphics mode, Spotfire S+ 8 enhanced graphics commands fail with the same error as would be present in S-PLUS 7.

To re-enable the enhanced graphics features:

```
> use.legacy.graphics(F)
```

For more information, see the `use.legacy.graphics` online help.

Using Device-Specific Palettes

By default, Spotfire S+ converts integers for color values to RGB values on a global basis, according to the values returned by the `palette()` and `image.palette()` functions. In this mode, the same RGB values are used for all devices. For more information, see the section Global Color Palette (page 9).

The `use.device.palette` function allows you to disable the global color palette functionality while running Spotfire S+ in the default graphics mode. That way, you can continue using the other enhanced graphics features, but have RGB conversion done on a device-specific basis.

To switch to device-specific palette mode, set the `use.device.palette` function to `TRUE` as follows:

```
> use.device.palette(T)
```

To use the palette settings available in S-PLUS prior to version 8.0, you must set `use.device.palette(T)`. For example, to produce black and white graphics as follows:

```
graphsheat(color.style = "black and white")
```

you must first set `use.device.palette(T)`.

Notes
If <code>use.legacy.graphics</code> is TRUE, the global color palettes are unavailable and device-specific palettes are used regardless of the <code>use.device.palette</code> setting. The <code>use.device.palette</code> setting is meaningful <i>only</i> while <code>use.legacy.graphics</code> is FALSE.

To re-enable the global color palettes:

```
> use.device.palette(F)
```

For more information, see the `use.device.palette` online help.

TRADITIONAL GRAPHICS

2

Introduction	29
Getting Started with Simple Plots	30
Vector Data Objects	30
Mathematical Functions	31
Scatter Plots	32
Frequently Used Plotting Options	33
Plot Shape	33
Multiple Plot Layout	33
Titles	34
Axis Labels	35
Axis Limits	36
Logarithmic Axes	37
Plot Types	37
Line Types	40
Plotting Characters	41
Controlling Plotting Colors	42
Visualizing One-Dimensional Data	43
Bar Plots	43
Pie Charts	45
Dot Charts	46
Notes and Suggestions	48
Visualizing the Distribution of Data	49
Box Plots	49
Histograms	50
Density Plots	52
Quantile-Quantile Plots	54
Visualizing Three-Dimensional Data	57
Contour Plots	57
Perspective Plots	59
Image Plots	60

Visualizing Multidimensional Data	62
Scatterplot Matrices	62
Plotting Matrix Data	63
Star Plots	64
Faces	65
Interactively Adding Information to Your Plot	66
Identifying Plotted Points	66
Adding Straight Line Fits to a Scatter Plot	67
Adding New Data to the Current Plot	68
Adding Text to Your Plot	69
Customizing Your Graphics	72
Low-level Graphics Functions and Parameters	72
Setting and Viewing Graphics Parameters	74
Controlling Graphics Regions	79
The Outer Margin	80
Figure Margins	81
The Plot Region	82
Controlling Text and Symbols	83
Text and Symbol Size	83
Text Placement	84
Text Orientation	85
Text in Figure Margins	86
Plotting Symbols in Margins	88
Line Width	88
Controlling Axes	89
Enabling and Disabling Axes	89
Tick Marks and Axis Labels	89
Axis Style	93
Axis Boxes	94
Controlling Multiple Plots	95
Multiple Figures on One Page	95
Pausing Between Multiple Figures	97
Overlaying Figures	97
Adding Special Symbols to Plots	102
Arrows and Line Segments	102
Stars and Other Symbols	104
Custom Symbols	106
Traditional Graphics Summary	108
References	113

INTRODUCTION

Visualizing data is a powerful data analysis tool because it allows you to easily detect interesting features or structure in the data. This may lead you to immediate conclusions or guide you in building a statistical model for your data. This chapter shows you how to use Spotfire S+ to visualize your data.

The first section in this chapter, *Getting Started with Simple Plots* (page 30), shows you how to plot vector objects and scatter plots. Once you have read this first section, you will be ready to use any of the options described in the section *Frequently Used Plotting Options* (page 33). The options, which can be used with many Spotfire S+ graphics functions, control the features in a plot, including plot shape, multiple plot layout, titles, and axes.

The remaining sections of this chapter cover a range of plotting tasks, including:

- Creating presentation graphics such as bar plots, pie charts, and dot plots.
- Visualizing the distribution of your data.
- Interactively adding information to your plot.
- Using multiple active graphics devices.

We recommend that you read the first two sections carefully before proceeding to any of the other sections.

In addition to the graphics features described in this chapter, Spotfire S+ includes the Trellis graphics library. Trellis graphics feature additional functionality such as multipanel layouts and improved 3D rendering. See Chapter 3, *Traditional Trellis Graphics* for more information.

GETTING STARTED WITH SIMPLE PLOTS

This section helps you get started with Spotfire S+ graphics by using the function `plot` to make simple plots of your data. You can use the `plot` function to create graphs of vector data objects and mathematical functions. In addition, `plot` creates scatter plots of two-dimensional data.

Vector Data Objects

You can use `plot` to graphically display the values in a batch of numbers or observations. For example, you obtain a graph of the built-in vector object `car.gals` using `plot` as follows:

```
> plot(car.gals)
```

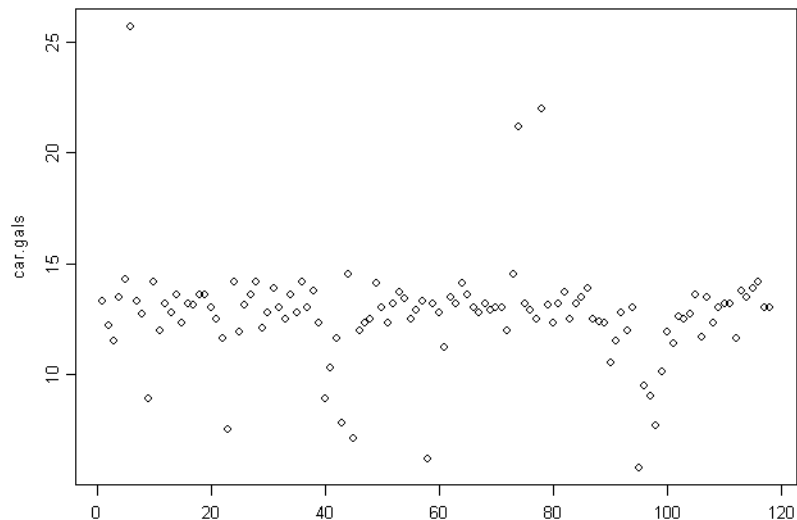


Figure 2.1: A scatter plot of `car.gals`, a single numeric vector.

The data are plotted as a set of isolated points. For each plotted point, the vertical axis location gives the data value and the horizontal axis location gives the observation number, or *index*.

If you have a vector x that is *complex*, `plot` plots the real part of x on the horizontal axis and the imaginary part on the vertical axis.

A set of points on the unit circle in the complex plane can be plotted as follows:

```
> unit.circle <- complex(argument =
+   seq(from = -pi, to = pi, length = 20))
> plot(unit.circle)
```

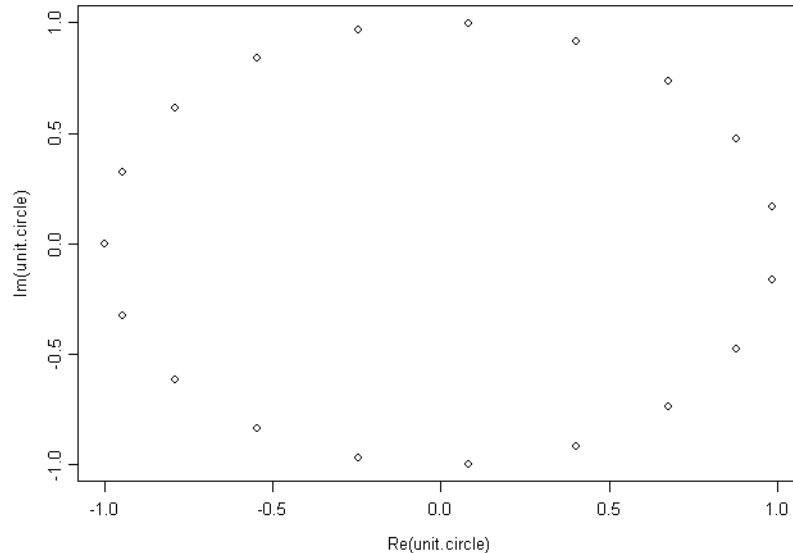


Figure 2.2: A scatter plot of a single complex vector.

Mathematical Functions

You can obtain solid line plots of mathematical functions by using the optional argument `type="l"` to `plot`. This option generates a plot with connected solid line segments rather than isolated points. The resulting plot is smooth, provided you choose a sufficiently dense set of plotting points.

For example, to plot the mathematical function in the equation:

$$y = f(x) = e^{-x/10} \cos(2x) \quad (2.1)$$

for x in the range $(0,20)$ first create a vector x with values ranging from 0 to 20 at intervals of 0.1. Next, compute the vector y by evaluating the function at each value in x , and then plot y against x :

```
> x <- seq(from = 0, to = 20, by = 0.1)
> y <- exp(-x/10) * cos(2*x)
> plot(x, y, type = "l")
```

The result is shown in Figure 2.3. For a rougher plot, use fewer points; for a smoother plot, use more.

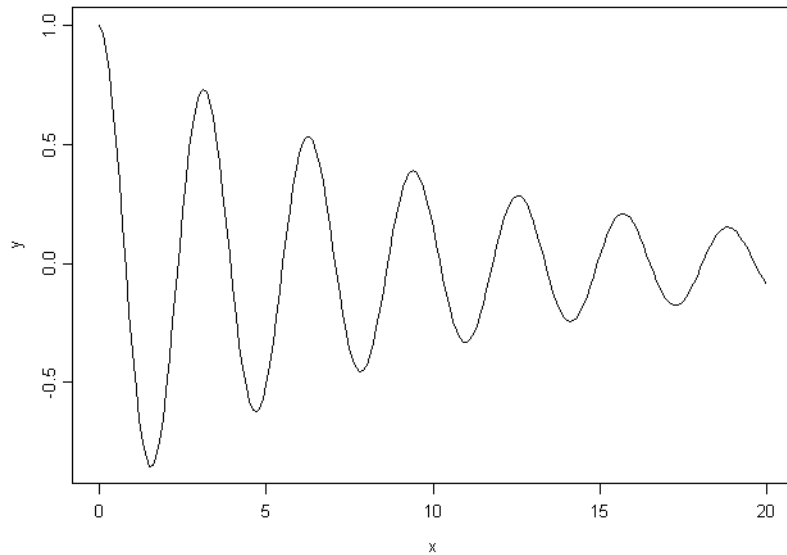


Figure 2.3: Plot of the function $\exp(-x/10)*\cos(2x)$.

Scatter Plots

Scatter plots reveal relationships between pairs of variables. You can create scatter plots in Spotfire S+ by applying the `plot` function to a pair of equal-length vectors, a matrix or data frame with two columns, or a list with components `x` and `y`. For example, to plot the built-in vectors `car.miles` versus `car.gals`, use this Spotfire S+ expression:

```
> plot(car.miles, car.gals)
```

When using `plot` with two vector arguments, the first argument is plotted along the horizontal axis and the second argument is plotted along the vertical axis.

If `x` is a matrix or data frame with two columns, use `plot(x)` to plot the second column versus the first. For example, you could combine the two vectors `car.miles` and `car.gals` into a matrix called `miles.gals` by using the function `cbind`:

```
> miles.gals <- cbind(car.miles, car.gals)
```

The following command produces the same graph as the command `plot(car.miles, car.gals)`:

```
> plot(miles.gals)
```

FREQUENTLY USED PLOTTING OPTIONS

This section tells you how to make plots in Spotfire S+ with one or more of a collection of frequently used options. These options include:

- Controlling plot shape and multiple plot layout
- Adding titles and axis labels
- Setting axis limits and specifying logarithmic axes
- Choosing plotting characters and line types
- Choosing plotting colors

Plot Shape

When you use a Spotfire S+ plotting function, the default shape of the box enclosing the plot is *rectangular*. Sometimes, you may prefer a *square* box around your plot. For example, a scatter plot is usually displayed as a square plot. You obtain a square box by using the global graphics parameter function `par` as follows:

```
> par(pty = "s")
```

All subsequent plots are made with a square box around the plot. If you want to return to making rectangular plots, use

```
> par(pty = "")
```

Here, the `pty` stands for *plot type* and the "s" is for *square*. However, you should think of `pty` as the *plot shape* parameter instead, to avoid confusion with a different parameter for plot type; see the section Plot Types (page 37) for more details.

Multiple Plot Layout

You may want to display more than one plot on your screen or on a single page of paper. To do so, you use the Spotfire S+ function `par` with the layout parameter `mfrow` to control the layout of the plots, as illustrated by the following example. In this example, you use `par` to set up a four-plot layout, with two rows of two plots each. Following the use of `par`, we create four simple plots with titles:

```
> par(mfrow = c(2,2))
> plot(1:10, 1:10, main = "Straight Line")
> hist(rnorm(50), main = "Histogram of Normal")
> qqnorm(rt(100,5), main = "Samples from t(5)")
```

```
> plot(density(rnorm(50)), main = "Normal Density")
```

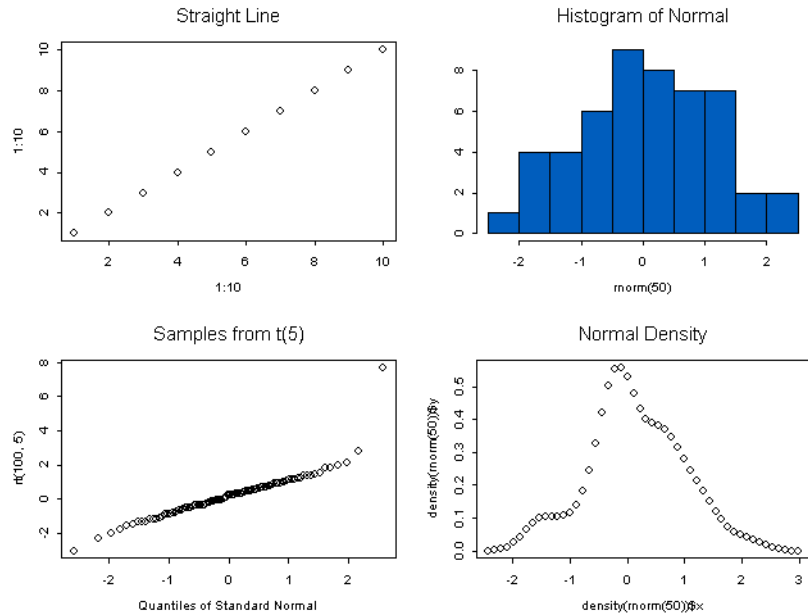


Figure 2.4: A four-plot layout using the `mfrow` parameter.

When you are ready to return to one plot per figure, use

```
> par(mfrow = c(1,1))
```

The `par` function is used to set many general graphics parameters. See the section [Setting and Viewing Graphics Parameters](#) (page 74) and the `par` help file for more information. The section [Controlling Multiple Plots](#) (page 95) contains details on using the `mfrow` parameter, and describes another method for creating multiple plots.

Titles

You can easily add titles to any Spotfire S+ plot. You can add a *main title*, which goes at the top of the plot, or a *subtitle*, which goes at the bottom of the plot. To place a main title on a plot, use the argument `main` to `plot`. For example:

```
> plot(car.gals, car.miles, main = "MILEAGE DATA")
```

To get a subtitle, use the `sub` argument:

```
> plot(car.gals, car.miles, sub = "Miles versus Gallons")
```

To get both a main title and a subtitle, use both arguments:

```
> plot(car.gals, car.miles, main = "MILEAGE DATA",
```

```
+ sub = "Miles versus Gallons")
```

The result is shown in Figure 2.5. Alternatively, you can add titles after creating the plot using the function `title`, as follows:

```
> plot(car.gals, car.miles)
> title(main="MILEAGE DATA", sub="Miles versus Gallons")
```

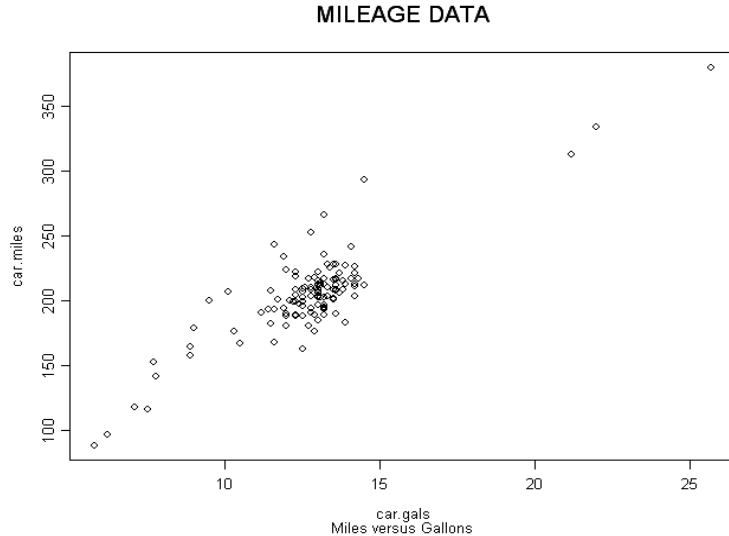


Figure 2.5: *A plot with main titles and subtitles.*

Axis Labels

Spotfire S+ provides default axis labels that are the names of the data objects passed as arguments to `plot`. However, you may want to use more descriptive axis labels in your graphs. For example, you may prefer “Gallons per Trip” and “Miles per Trip,” respectively, to “car.gals” and “car.miles.” To obtain your preferred labels, use the `xlab` and `ylab` arguments as follows:

```
> plot(car.gals, car.miles,
+ xlab = "Gallons per Trip", ylab = "Miles per Trip")
```

You can also suppress axis labels by using the arguments `xlab` and `ylab` with `""`, the empty string value. For example:

```
> plot(car.gals, car.miles, xlab = "", ylab = "")
```

This results in a plot with no axis labels. If desired, you can later add labels using the `title` function:

```
> title(xlab = "Gallons per Trip", ylab = "Miles per Trip")
```

Axis Limits

The limits of the x and y axes are set automatically by the Spotfire S+ plotting functions. However, you may wish to choose your own axis limits. This allows you to make room for adding text in the body of a plot, as described in the section Interactively Adding Information to Your Plot (page 66). For example, the command

```
> plot(co2)
```

automatically determines y axis limits of roughly 310 and 360, giving enough vertical space for the plot to fit in the box. You can include more vertical or horizontal space by using the optional arguments `ylim` and `xlim`. Spotfire S+ then rounds your specified axis limits to sensible values. For example, to get y axis limits of 300 and 370, use:

```
> plot(co2, ylim = c(300,370))
```

Using the `xlim` parameter, you can change the x axis limits as well:

```
> plot(co2, xlim = c(1955,1995))
```

You can also use both `xlim` and `ylim` at the same time:

```
> plot(co2, xlim = c(1955,1995), ylim = c(300,370))
```

You may also want to set axis limits when you create multiple plots, as described in the section Multiple Plot Layout (page 33). For example, after creating one plot, you may wish to make the x and y axis limits the same for all of the plots in the set. You can do so by using the `par` function as follows:

```
> par(xaxs = "d", yaxs = "d")
```

If you want to control the limits of only one of the axes, drop one of the two arguments as appropriate. Using `xaxs="d"` and `yaxs="d"` sets all axis limits to the values for the most recent plot in a sequence. If those limits are not wide enough for all plots in the sequence, points outside the limits are not plotted and you receive the message `Points out of bounds`. To avoid this error, first create all plots in the usual way without specifying axis limits, to find out which plot has the largest range of values. Then, create your first plot using `xlim` and `ylim` with values determined by the largest range. Next, set the axes with `xaxs="d"` and `yaxs="d"` as described above.

To return to the usual default state, in which each plot determines its own limits in a multiple plot layout, use

```
> par(xaxs = "", yaxs = "")
```

The change takes effect on the next page of figures.

Logarithmic Axes

Often, a data set you are interested in does not reveal much detail when graphed on ordinary axes. This is particularly true when many of the data points bunch up at small values, making it difficult to see any potentially interesting structure in the data. Such data sets yield more informative plots if you graph them using a *logarithmic scale* for one or both of the axes.

To draw the horizontal axis of a plot on a logarithmic scale, use the argument `log="x"` in the call to the graphics function. Similarly for the vertical axis, use `log="y"` to draw the vertical axis on a logarithmic scale. To put both axes on logarithmic scales, use the option `log="xy"`.

Plot Types

In Spotfire S+, you can plot data in any of the following ways:

- As points
- As connected straight line segments
- As both points and lines, where the points are isolated
- As overstruck points and lines, where the points are not isolated
- With a vertical line for each data point (*high-density plot*)
- As a stairstep plot
- As an empty plot that has axes and labels, but no data plotted

The method used for plotting data on a graph is called the graph's *plot type*. For example, scatter plots typically use the first plot type (`type="p"`), while time series plots typically use the second (`type="l"`). In this section, we provide examples of each of these plot types.

You choose your plot type through the optional argument `type`. The possible values for this argument are given in Table 2.1, and correspond to the choices listed above.

Table 2.1: Possible values of the *plot type* argument.

Setting	Plot type
<code>type="p"</code>	points
<code>type="l"</code>	lines
<code>type="b"</code>	both points and lines
<code>type="o"</code>	lines with points overstruck
<code>type="h"</code>	high-density plot
<code>type="s"</code>	stairstep plot
<code>type="n"</code>	no data plotted

Different graphics functions have different default values for the `type` argument. For example, `plot` and `matplot` use `type="p"`, while `ts.plot` uses `type="l"`. Although you can use any of the plot types with any plotting function, some combinations of plot functions and plot types may result in an ineffective display of your data.

The option `type="n"` is useful for obtaining precise control over axis limits and box line types. For example, you might want the axes and labels displayed in one color, and the data plotted in another. The following commands show how to do this for arbitrary data vectors `x` and `y`.

```
> plot(x, y, type = "n")
> points(x, y, col = "darkslategray")
```

Figure 2.6 shows the different plot types for the built-in data set `car.miles`, as produced by the following commands:

```
> plot(car.miles)
> plot(car.miles, type = "l")
> plot(car.miles, type = "b")
> plot(car.miles, type = "o")
> plot(car.miles, type = "h")
> plot(car.miles, type = "s")
```

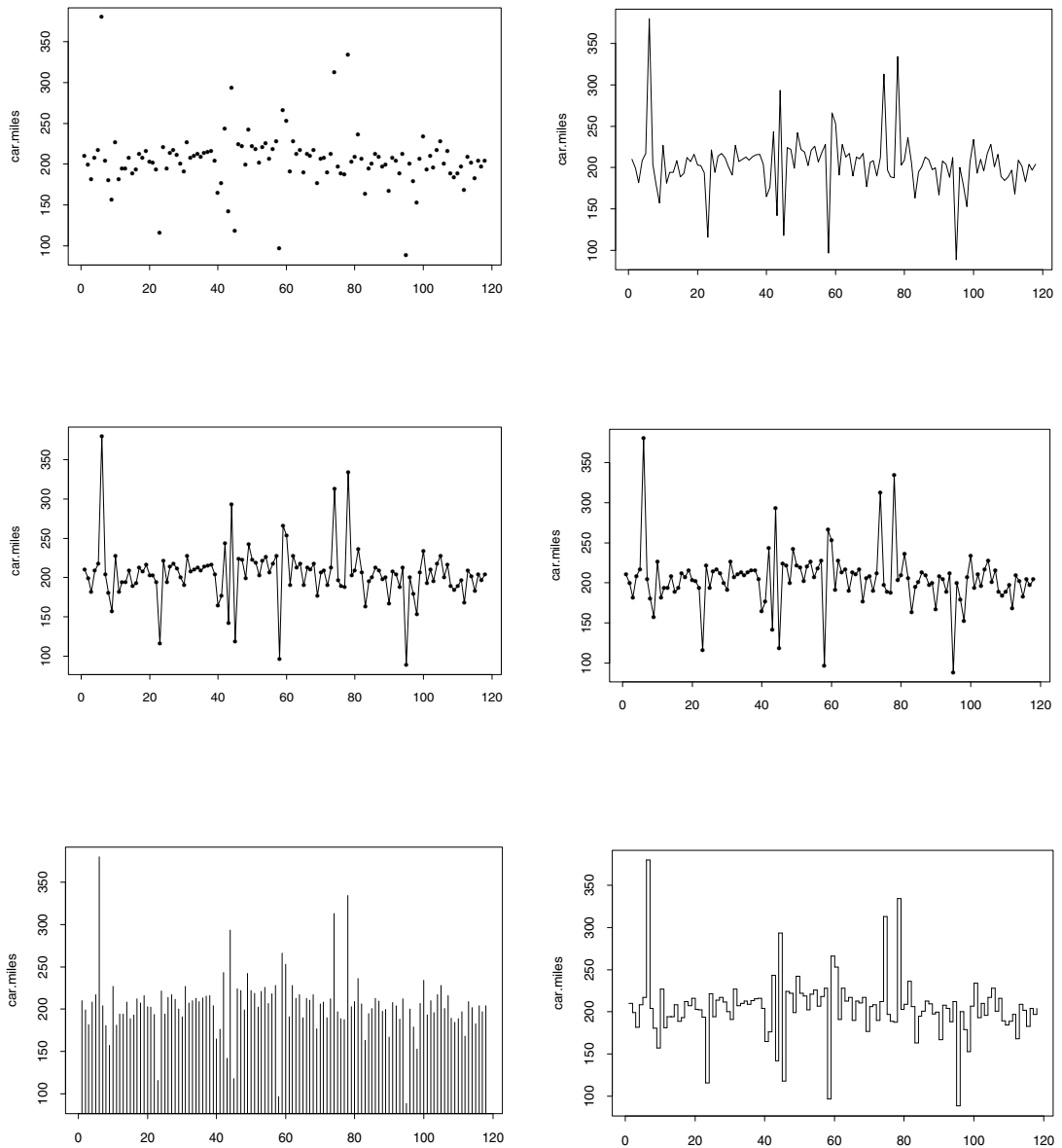



Figure 2.6: *Plot types for the function `plot`. Top row: points and lines. Second row: both points and lines, and lines with points overstruck. Third row: high density plot and staircase plot.*

Line Types

When your plot involves multiple lines, you can choose specific *line types* to distinguish between them. By default, the first line on a graph is a solid line. If you prefer, you can use the `lty` argument to specify a different line type. On the most commonly used graphics devices, there are eight distinct line types, as shown in Figure 2.7.

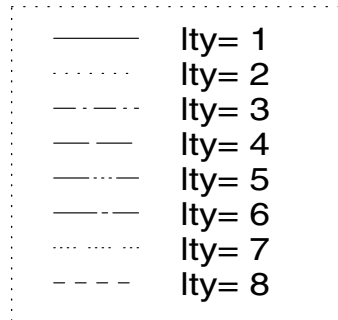


Figure 2.7: Line types from the `lty` parameter.

In a device with eight line types, if you specify a value higher than eight for the `lty` argument, Spotfire S+ produces the line type corresponding to the *remainder* on division by the number of line types. For example, if you specify `lty=26` on the `graphsheat` device (Windows) or the `motif` device (UNIX), Spotfire S+ produces the line type shown as `lty=2`.

Notes

The value of `lty` can be an integer or a vector. For example, you can plot the time series `halibut$cpue` using a dotted line as follows:

```
> plot(halibut$cpue, type = "l", lty = 2)
```

The following example illustrates using vectors of `col`, `lty`, and `lwd` values:

```
> plot(1:10, type="n")
> lines(1:10, rep(c(1,2),5), col=c("red", "green", "blue"),
+      lty=1:3, lwd=c(1,3,5))
```

For more information, see the section [Vectorized Graphics Parameters](#) (page 21) and the [par](#) online help.

Plotting Characters

When your plot type involves data points, you can choose the *plotting character* for the points. By default, the plotting character is usually an open circle, depending on your graphics device and the plotting function you use. For example, the default plotting character for `matplot` is the number 1, because `matplot` is often used to simultaneously plot more than one time series or vector. In such cases, a plotting character is needed for each *time series* or vector to be plotted, and the default characters are the integers 1, 2, 3, etc.

You can choose alternative plotting characters for your data points by using the optional argument `pch`. Any printing character can be used as the value of `pch`, as long as it is enclosed in quotes. For example:

```
> plot(halibut$biomass, pch = "B")
```

You can also choose any one of a range of plotting symbols by using `pch=n`, where n is an integer. The symbol corresponding to each integer is shown in Figure 2.8. For example, to plot the series `halibut$biomass` using a filled triangle as the plotting character, type the following command:

```
> plot(halibut$biomass, pch = 17)
```

□	0	○	1	△	2
+	3	×	4	◇	5
▽	6	⊠	7	✱	8
⊕	9	⊕	10	⊗	11
⊞	12	⊗	13	⊠	14
■	15	●	16	▲	17
◆	18				

Figure 2.8: Plotting symbols from the `pch` parameter.

Note

The plot function, and functions that add individual points to a plot, can accept a vector of values for the pch parameter. For example:

```
plot(halibut$biomass, pch = rep(0:2,each=20))
```

For more information, see the section Vectorized Graphics Parameters (page 21).

**Controlling
Plotting Colors**

To specify the color in which your graphics are plotted, use the optional col parameter. This parameter is useful when you need to distinguish between sets of overlaid data, as the two commands below illustrate.

```
> plot(co2)
> lines(smooth(co2), col = "olive")
```

Notes

For information about the colors available through the col parameter, see the section Color Specification (page 3).

For information about using a vector of values with col and other graphics parameters, see the section Vectorized Graphics Parameters (page 21).

For information about using separate graphics arguments to control the various individual elements of a plot (e.g., col.axis, col.lab, col.main, col.sub, fg, and bg) see the section Additional Graphics Arguments (page 17).

For information about setting graphics color schemes in the Spotfire S+ GUI, refer to the online help.

VISUALIZING ONE-DIMENSIONAL DATA

Bar plots and pie charts are familiar methods of graphically displaying data for oral presentations, reports, and publications. In this section, we show you how to use Spotfire S+ to make these plots. We also show you how to make another type of one-dimensional plot called a *dot chart*. Dot charts are less widely known than the more familiar bar plots and pie charts, but they are often more useful.

We illustrate each of the three plots with the following 5 x 3 matrix `digits`:

```
> digits <- matrix(
+   c(20,15,30,16,17,30,24,16,17,21,24,20,19,13,28),
+   nrow = 5, byrow = T)

> dimnames(digits) <- list(
+   paste("digit", 1:5, sep=" "),
+   paste("sample", 1:3, sep=" "))

> digits
```

	sample 1	sample 2	sample 3
digit 1	20	15	30
digit 2	16	17	30
digit 3	24	16	17
digit 4	21	24	20
digit 5	19	13	28

For convenience in the examples below, we extract the row and column labels from the matrix and store them in separate objects:

```
> digit.names <- dimnames(digits)[[1]]
> sample.names <- dimnames(digits)[[2]]
```

Bar Plots

The function `barplot` is a flexible function for making bar plots. The simplest use of `barplot` is with a vector or a single column from a matrix. For example, calling `barplot` with the first column of `digits` gives the result shown in Figure 2.9.

```
> barplot(digits[,1], names = digit.names)
```

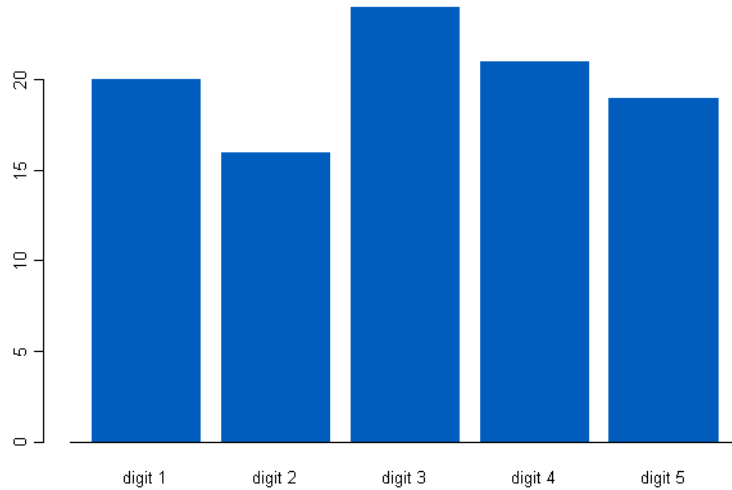


Figure 2.9: A bar plot of the *digits* data.

In the figure, the height of each bar is the value in the corresponding component of the vector or matrix column; in most instances, the values represent counts of some sort. The `barplot` function can also be used in a more powerful way to create a bar plot of an entire data matrix. In this case, each bar corresponds to a column of the matrix and represents a sample. Each bar is divided into a number of blocks representing the values, with different shadings in each of the blocks. You can see this with the `digits` data as follows:

```
> barplot(digits,
+   angle = seq(from = 45, to = 135, length = 5),
+   density = 16, names = sample.names)
```

Our value for the optional argument `angle` establishes five angles for the shading fill for each of the five blocks in each bar, with the angles equally spaced between 45 degrees and 135 degrees. Setting the optional argument `density` to 16 causes the shading fill lines to have a density of 16 lines per inch. If you want the density of the shading fill lines to vary cyclically, you need to set `density` with a vector value; the vector is of length five for the `digits` data. For example:

```
> barplot(digits,
+   angle = seq(from = 45, to = 135, length = 5),
+   density = (1:5)*5, names = sample.names)
```

To produce a legend that associates a name to each block of bars, use the optional `legend` argument with an appropriate character vector as its value. For the `digits` example, use `legend=digit.names` to associate a digit name with each of the blocks in the bars:

```
> barplot(digits, angle = c(45,135), density = (1:5)*5,  
+         names = sample.names, legend = digit.names,  
+         ylim = c(0,270))
```

To make room for the legend, you usually need to increase the range of the vertical axis, so we use `ylim=c(0,270)`. You can obtain greater flexibility for the positioning of the legend by using the function `legend` after you have made your bar plot, rather than relying on the automatic positioning from the `legend` argument. See the section *Adding legends* (page 70) for more information.

Many other options are available to you as arguments to `barplot`; see the help file for complete details.

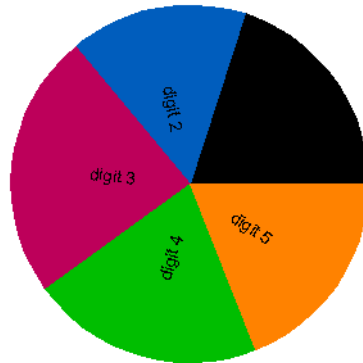
Pie Charts

You can make pie charts with the function `pie`. For example, you can display the first sample of the `digits` data as a pie chart and add the subtitle “sample 1” by using `pie` as follows:

```
> pie(digits[,1], names = digit.names, sub = "sample 1")
```

The result is shown in Figure 2.10. As an alternative, try replacing `digits[,1]` by `digits[,2]` or `digits[,3]`, and replace “sample 1” by “sample 2” or “sample 3”, respectively.

Several other options are available with the `pie` function; see the help file for complete details.



sample 1

Figure 2.10: *A pie chart of the digits data.*

Dot Charts

The dot chart was first described by Cleveland (1985) as an alternative to bar plots and pie charts. The dot chart displays the same information as a bar plot or pie chart, but in a form that is often easier to grasp. In particular, the dot chart reduces most data comparisons to straightforward length comparisons on a common scale.

In Spotfire S+, use the function `dotchart` to create dot plots of your data. The simplest use of `dotchart` is analogous to that of `barplot`. You can see this by calling `dotchart` with the first column of the `digits` matrix:

```
> dotchart(digits[,1], labels = digit.names)
```

The result is displayed in Figure 2.11.

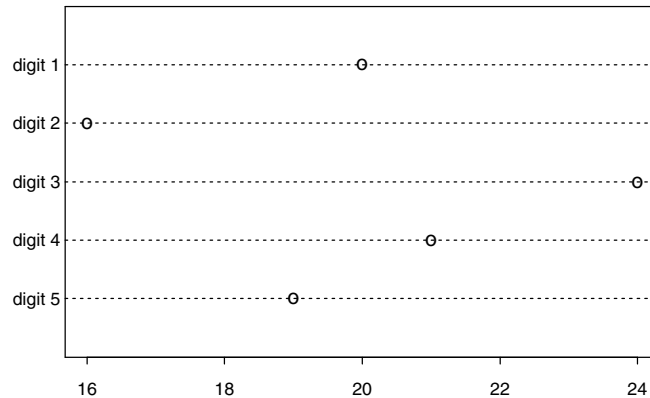


Figure 2.11: *A dot chart of the digits data.*

To obtain a display of all the data in the `digits` matrix, use the following command:

```
> dotchart(digits, labels = digit.names)
```

Alternatively, you can also use the following command:

```
> dotchart(t(digits), labels = sample.names)
```

The argument `t(digits)` uses the function `t` to *transpose* the `digits` matrix (i.e., to interchange the rows and columns of `digits`).

To obtain a plot of `digits` with both the sample labels and the digit labels, you need to create a grouping variable to use as an additional argument. For example, if you wish to use the sample number as the grouping variable, use the `factor` function to create the variable `sample.fac` as follows:

```
> sample.fac <- factor(col(digits), labels = sample.names)
```

You can then use this factor object as the `groups` argument to `dotchart`:

```
> dotchart(digits, labels=digit.names, groups=sample.fac)
```

For more information on factor objects, see Chapter 4, Data Objects in the *Programmer's Guide*.

Several other options are available with the `dotchart` function; see the help file for complete details.

Notes and Suggestions

A pie chart shows the share of individual values in a variable, relative to the sum total of all the values. Pie charts display the same information as bar plots and dot charts, but can be more difficult to interpret. This is because the size of a pie wedge is relative to a sum, and does not directly reflect the magnitude of the data value. Because of this, pie charts are most useful when the emphasis is on an individual item's relation to the whole; in these cases, the sizes of the pie wedges are naturally interpreted as percentages. When such an emphasis is not the primary point of the graphic, a bar plot or a dot chart is preferred.

In some cases, bar plots also introduce perceptual ambiguities; this is particularly evident in divided bar charts. For these reasons, we recommend dot charts for general displays of one-dimensional data.

VISUALIZING THE DISTRIBUTION OF DATA

For any data set you need to analyze, you should try to get a visual picture of the shape of its distribution. The distribution shape is readily visualized from such familiar plots as box plots, histograms, and density plots. Less familiar but equally useful are *quantile-quantile plots*, or qqplots. In this section, we show you how to use Spotfire S+ functions to make these kinds of plots.

Box Plots

A box plot is a simple graphical representation showing the center and spread of a distribution, along with a display of unusually deviant data points called *outliers*. To create a box plot in Spotfire S+, use the `boxplot` function:

```
> boxplot(corn.rain)
```

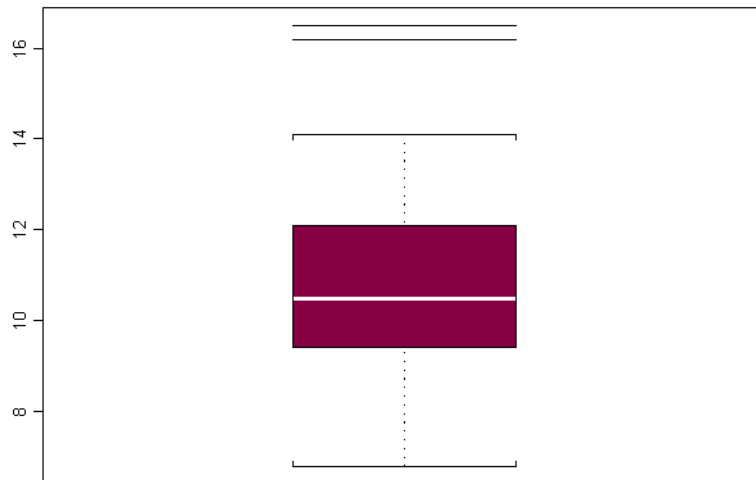


Figure 2.12: A box plot of the `corn.rain` data.

The horizontal line in the interior of the box is located at the median of the data. This estimates the center of the distribution for the data.

The boxplot function uses *hinges*, as originally defined by Tukey, for the lower and upper limits of the box. The hinges are the median value of each half of the data, where the overall median defines the halves. Hinges are similar to quartiles. The difference between hinges

and quartiles is that the depth (that is, the distance from the lower and upper limits of the data) of the hinges is calculated from the depth of the median. Hinges often lie slightly closer to the median than do the quartiles.

Note

The difference between hinges and quartiles is usually quite small. If you are interested in quantiles, you should use the `quantile` or `summary.default` functions instead of the `stats` component returned by `boxplot`.

Points beyond the whiskers are considered outliers and are drawn individually, indicated as horizontal lines. Supplying the optional argument `range=0` to `boxplot` forces the whiskers to span the full data range; any positive value of `range` multiplies the standard span by that amount. The *standard span* is 1.5 times the upper hinge minus the lower hinge.

For data having a Gaussian distribution, approximately 99.3% of the data falls inside the whiskers of a box plot, given the standard span. In the `corn.rain` example, the two horizontal lines at the top of the graph in Figure 2.12 represent outliers.

Box plots provide a very powerful method for visualizing the rough distributional shape of two or more samples of data. For example, to compare the distributions of the New Jersey lottery payoffs in each of three different years, call `boxplot` with the built-in data vectors `lottery.payoff`, `lottery2.payoff`, and `lottery3.payoff` as follows:

```
> boxplot(lottery.payoff, lottery2.payoff, lottery3.payoff)
```

You can modify the style of your box plots using optional arguments to the `boxplot` function; see the help file for complete details.

Histograms

A histogram shows the number of data points that fall in each of a number of intervals. You can create histograms in Spotfire S+ with the `hist` function:

```
> hist(corn.rain)
```

The simple histogram displayed spans the range of the data; the smallest data value falls in the leftmost interval and the largest data point falls in the rightmost interval. Notice that the histogram gives

you an indication of the relative density of the data points along the horizontal axis. In the `corn.rain` example, there are 10 data points in the interval (8,10) and only one data point in the interval (14,16).

The number of intervals produced by `hist` is determined automatically to balance the trade-off between obtaining smoothness and preserving detail. However, no automatic rule is completely satisfactory. Thus, `hist` allows you to choose the number of intervals yourself through the optional argument `nclass`. Choosing a large number of intervals produces a rougher histogram with more detail, and choosing a small number produces a smoother histogram with less detail. For example, the command

```
> hist(corn.rain, nclass = 15)
```

gives a rougher but more detailed histogram than the one produced by `hist(corn.rain)`.

You can also use `hist` to specify the number of intervals and their locations. You do this through the optional argument `breaks`, by specifying a numeric vector containing the interval boundary points. The length of this vector is one greater than the number of intervals you want. For example, to specify 12 intervals for the `corn.rain` histogram with interval boundaries at the integers 6 through 18, use

```
> hist(corn.rain, breaks = 6:18)
```

The result is shown in Figure 2.13. Many other options are available with `hist`, and they include many of the arguments to `barplot`. See the help files for `hist` and `barplot` for complete details.

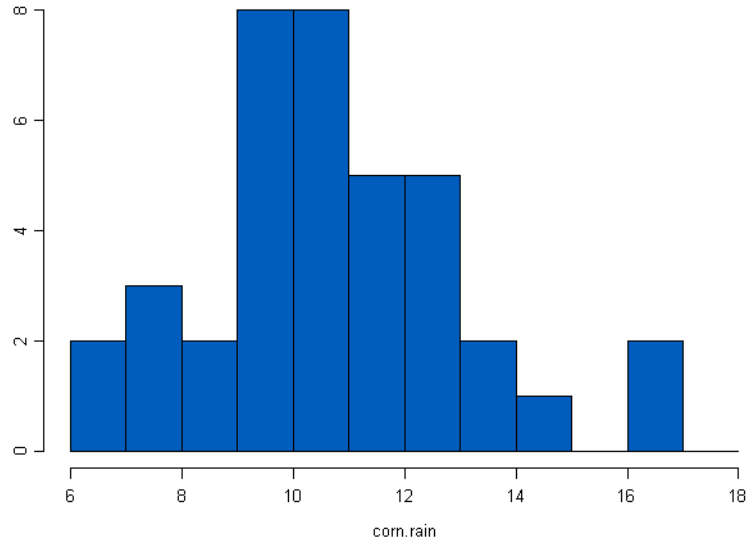


Figure 2.13: A histogram of the `corn.rain` data with specified break points.

Density Plots

A histogram for continuous numeric data is a rough estimate of a smooth underlying *density curve*, which gives the relative frequency with which the data fall in different intervals. The underlying density curve, formally called a *probability density function*, allows you to compute the probability that your data fall in any interval. Thus, you may prefer a smooth estimate of the density to a rough histogram estimate.

To obtain smooth density estimates in Spotfire S+, use `plot` with the function `density`. The optional argument `width` to `density` controls the smoothness of the plot. For example, Figure 2.14 shows the plots displayed by the following two commands:

```
> plot(density(car.gals), type = "l")  
> plot(density(car.gals, width = 1), type = "l")
```

The default value for `width` results in a smooth density estimate in the tails, whereas the choice `width=1` produces a rougher estimate. In general, larger `width` values result in smoother plots but may obscure

local details of the density. Smaller width values highlight local details better, but may also highlight random effects. See Silverman (1986) or Venables and Ripley (1999) for a discussion of the issues involved in choosing a width parameter.

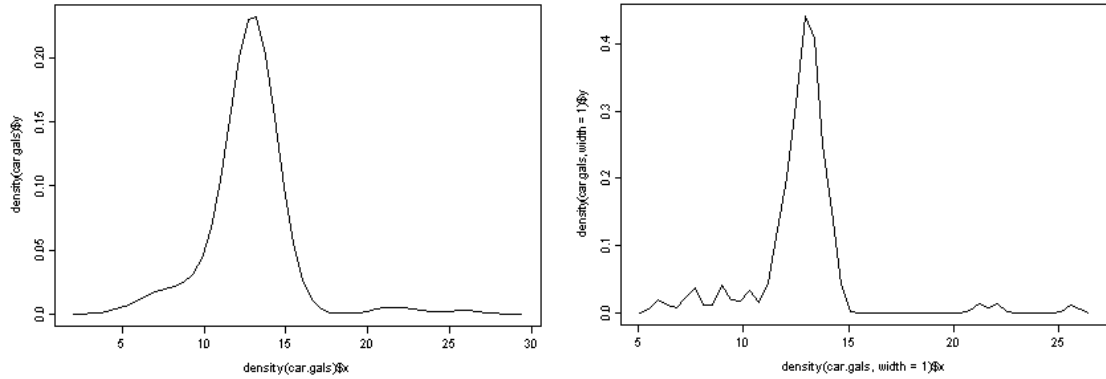


Figure 2.14: Probability density plots of the *car.gals* data.

Quantile-Quantile Plots

A *quantile-quantile plot*, or qqplot, is a plot of one set of quantiles against another. There are two main forms of the qqplot. The most frequently used form checks whether a data set comes from a particular hypothesized distribution shape. In this case, one set of quantiles consists of the ordered set of data values (which are, in fact, quantiles for the empirical distribution for the data). The other set consists of quantiles for the hypothesized distribution. If the points in this plot cluster along a straight line, the data set likely has the hypothesized distribution.

The second form of qqplot is used when you want to find out whether two data sets have the same distribution shape. In this case, both sets of quantiles simply consist of ordered data values. If the points in this plot cluster along a straight line, the two data sets likely have the same distribution shape.

QQplots for Checking Distribution Shape

To produce the first type of qqplot when your hypothesized distribution is normal (Gaussian), use the function `qqnorm`:

```
> qqnorm(car.gals)
> qqline(car.gals)
```

The result is shown in Figure 2.15. The `qqline` function computes and draws a robust straight line fit that is not greatly influenced by outliers.

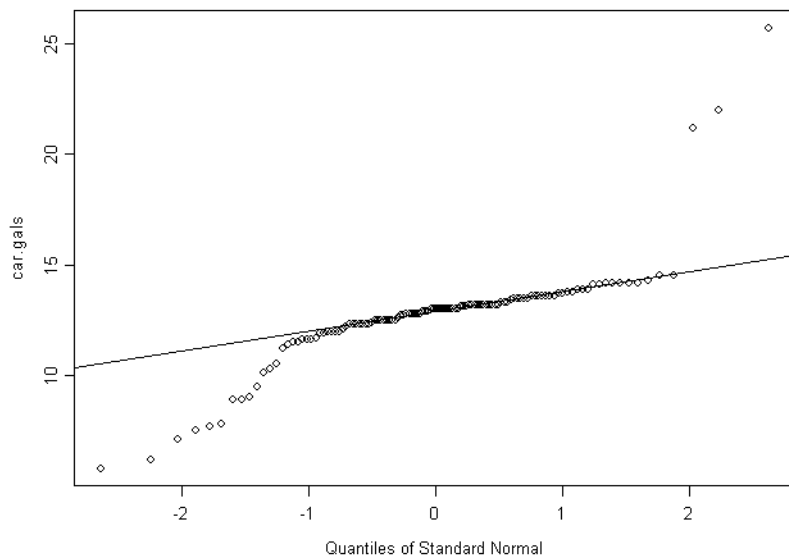


Figure 2.15: A *qqnorm* plot of the *car.gals* data.

You can create qqplots to check whether your data come from any of a number of distributions. To do so, you need to write a simple Spotfire S+ function for your hypothesized distribution; we illustrate this idea for the uniform distribution.

Create the function `qqunif` as follows:

```
> qqunif <- function(x) {
+   plot(qunif(ppoints(x)), sort(x))
+ }
```

The function `qunif` computes quantiles for the uniform distribution at the probability values returned by `ppoints`, and `sort` orders the data. If n is the length of x , the probabilities computed by `ppoints` satisfy $p[x_i] = (i - 0.5)/n$. We can now call `qqunif` to check whether the `car.gals` data comes from a uniform distribution:

```
> qqunif(car.gals)
```

You can create qqplots for other hypothesized distributions by replacing `qunif` in the definition of `qqunif` by one of the functions from Table 2.2.

Table 2.2: *Distributions for qqplots.*

Function	Distribution	Required Arguments	Optional Arguments	Defaults
<code>qbeta</code>	beta	<code>shape1, shape2</code>	none	
<code>qcauchy</code>	Cauchy	none	<code>location, scale</code>	<code>0, 1</code>
<code>qchisq</code>	chi-square	<code>df</code>	none	
<code>qexp</code>	exponential	none	<code>rate</code>	<code>1</code>
<code>qf</code>	F	<code>df1, df2</code>	none	
<code>qgamma</code>	Gamma	<code>shape</code>	none	
<code>qlnorm</code>	log-normal	none	<code>mean, sd</code>	<code>0, 1</code>

Table 2.2: Distributions for qqplots.

Function	Distribution	Required Arguments	Optional Arguments	Defaults
<code>qnorm</code>	normal	none	<code>mean, sd</code>	<code>0, 1</code>
<code>qt</code>	Student's t	<code>df</code>	none	
<code>qunif</code>	uniform	none	<code>min, max</code>	<code>0, 1</code>

Note

For distribution functions requiring a parameter argument, your qqplot function must accept it. For example, `qqchisq` must accept the required `df` argument as follows:

```
> qqchisq <- function(x,df) { plot(qchisq(ppoints(x), df), sort(x)) }
```

QQplots for Comparing Two Sets of Data

When you want to check whether two sets of data have the same distribution, use the function `qqplot`. If the data sets have the same number of observations, `qqplot` plots the ordered data values of one versus the ordered data values of the other. If the two data sets do not have the same number of observations, the ordered data values for one set are plotted against interpolates of the ordered data values of the other set.

For example, to compare the distributions of the two New Jersey lottery data vectors `lottery.payoff` and `lottery3.payoff`, use the following expression:

```
> qqplot(lottery.payoff, lottery3.payoff)
```

VISUALIZING THREE-DIMENSIONAL DATA

Many types of data are usefully viewed as *surfaces* generated by functions of two variables. Familiar examples are meteorological data, topographic data, and other data gathered by geographical location.

Spotfire S+ provides three functions for viewing such data. The simplest, `contour`, represents the surface as a set of contour plot lines on a grid representing the other two variables. The perspective plot, `persp`, creates a perspective plot with hidden line removal. The `image` function plots the surface as a color or grayscale variation on the base grid.

All three functions require similar input: a vector of x coordinates, a vector of y coordinates, and a length x by length y matrix of z values. In many cases, these arguments are all supplied by a single list, such as the output of the `interp` function. The `interp` function *interpolates* the value of the third variable onto an evenly spaced grid of the first two variables. For example, the built-in data set `ozone` contains two objects: `ozone.xy`, a list of latitudes and longitudes for each observation site, and `ozone.median`, a vector of the medians of daily maxima ozone concentrations at all sites. To create a contour or perspective plot, we can use `interp` to interpolate the data as follows:

```
> ozone.fit <- interp(ozone.xy$x, ozone.xy$y, ozone.median)
```

We use the `ozone.fit` object in examples throughout this section.

For `contour`, `persp`, and `image` you can also provide a single matrix argument, which is interpreted as the z matrix. The two functions then automatically generate the x vector `1:nrow(z)` and the y vector `1:ncol(z)`. See the `persp` and `contour` help files for more information.

Contour Plots

To generate a contour plot, use the `contour` function. For example, the built-in data set `switzerland` contains elevation data for Switzerland. The following command produces the plot shown in Figure 2.16:

```
> contour(switzerland)
```

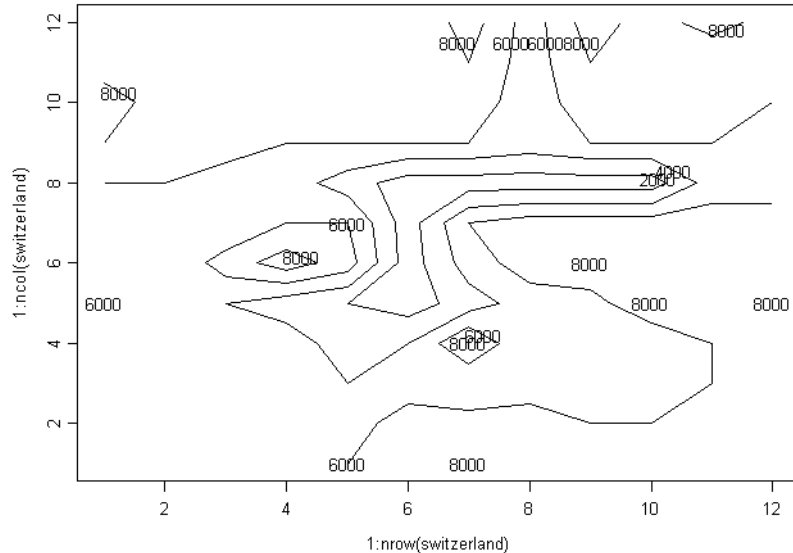


Figure 2.16: *Contour plot of Switzerland.*

By default, `contour` draws contour lines for each of five levels, and labels each one. You can change the number of levels with either `nlevels` or `levels`. The `nlevels` argument specifies the approximate number of contour intervals desired, while the `levels` argument specifies a vector of heights for the contour lines.

You control the size of the labels for the contour lines with the `labex` argument. Specify the size to `labex` as a relative value to the current axis-label font, so that `labex=1` (the default) yields labels that are the same size as the axis labels. Setting `labex=0` gives you unlabeled contour lines.

For example, to view a voice spectrogram for the word “five,” use `contour` on the built-in data object `voice.five`. Because `voice.five` generates many contour lines, we suppress the labels with `labex=0`:

```
> contour(voice.five, labex = 0)
```

If you have an equal number of observations for each of three variables, you can use `interp` to generate interpolated values for z on an equally-spaced xy grid. For example, to create a contour plot of the ozone data, you can use `contour` with the `ozone.fit` object as follows:

```
> contour(ozone.fit)
```

Perspective Plots

Perspective plots give a three-dimensional view of data in the form of a matrix of heights on an evenly spaced grid. The heights are connected by line segments to produce the familiar mesh appearance of such plots.

As a simple example, consider the voice spectrogram for the word “five” stored in the data set `voice.five`. The contour plot we created in the previous section is difficult to interpret because the number of contour lines forced us to omit the height labels. Had we included the labels, the clutter would have made the graph unreadable.

The perspective plot in Figure 2.17 gives a much clearer view of how the spectrogram varies. To create the plot, use the following Spotfire S+ expression:

```
> persp(voice.five)
```

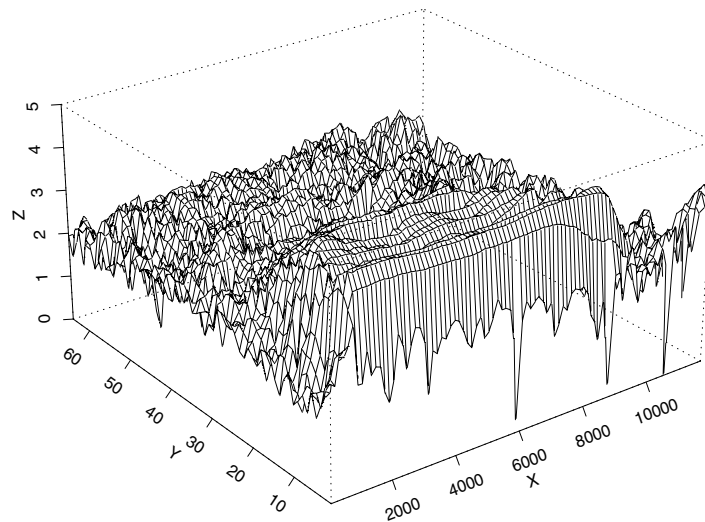


Figure 2.17: *Perspective plot of a voice spectrogram.*

You can modify the perspective by choosing a different “eye” location. You do this with the `eye` argument to `persp`. By default, the eye is located at $c(-6, -8, 5)$ times the range of the x , y , and z values. For example, to look at the voice data from the side opposite of what is shown in Figure 2.17, we could use the following command:

```
> persp(voice.five, eye = c(72000,350,30))
```

If you have an equal number of observations for each of three variables, you can use `interp` to generate interpolated values for z on an equally-spaced xy grid. For example, to create a perspective plot of the ozone data, you can use `persp` with the `ozone.fit` object as follows:

```
> persp(ozone.fit)
```

Warning

Converting a `persp` plot to individual objects can take a considerable amount of time. For this reason, we recommend against converting `persp` plots to editable graphics.

Image Plots

An image plot is a two-dimensional plot that represents three-dimensional data as shades of color or grayscale. You can produce image plots with the `image` function:

```
> image(voice.five)
```

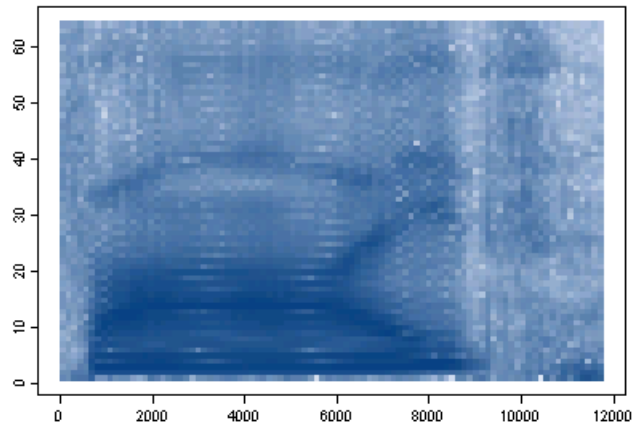


Figure 2.18: *Image plot of the voice spectrogram.*

A more conventional use of `image` is to produce images of topological data, as in the following example:

```
> image(pugetN)
```

The data set `pugetN` contains elevations in and around Puget Sound. It is not part of the standard Spotfire S+ distribution.

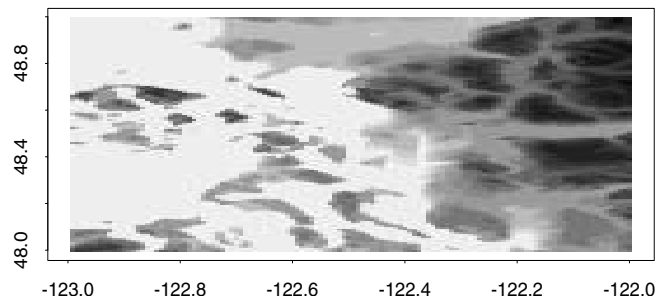


Figure 2.19: *Image plot of Puget Sound.*

If you have an equal number of observations for each of three variables, you can use `interp` to generate interpolated values for z on an equally-spaced xy grid. For example, to create an image plot of the ozone data, you can use `image` with the `ozone.fit` object as follows:

```
> image(ozone.fit)
```

VISUALIZING MULTIDIMENSIONAL DATA

For data with three or more variables, many methods of graphical visualization have been developed. Some of these are highly interactive and take full advantage of the power of personal computers. The following sections describe how to use Spotfire S+ functions to analyze multidimensional data. In particular, we describe several methods for static data visualization that are widely considered useful: scatterplot matrices, matrix plots, star plots, and Chernoff's faces.

Scatterplot Matrices

A *scatterplot matrix* is an array of pairwise scatter plots showing the relationship between any pair of variables in a multivariate data set. To produce a static scatterplot matrix in Spotfire S+, use the `pairs` function with an appropriate data object as its argument. For example, the following Spotfire S+ expression generates a scatterplot matrix of the built-in data set `longley.x`:

```
> pairs(longley.x)
```

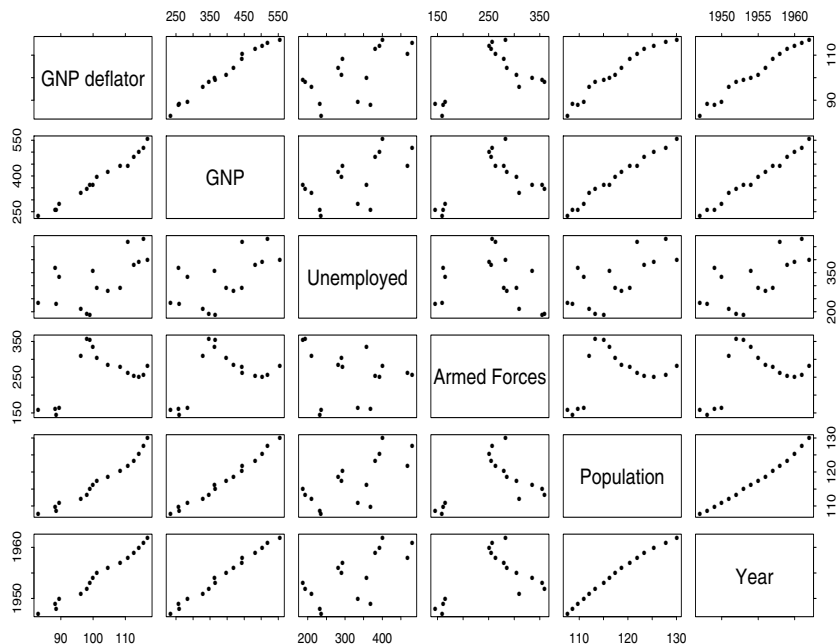


Figure 2.20: A scatterplot matrix of the `longley.x` data.

Plotting Matrix Data

For visualizing several vector data objects, or for visualizing some kinds of multivariate data, you can use the function `matplot`. This function plots columns of one matrix against columns of another. For example, Spotfire S+ contains a multivariate data set named `iris`. The `iris` data is in the form of a data *array*, which is a generalized matrix. Let's extract two particular 50x3 matrices from the `iris` array:

```
> pet.length <- iris[,3,]
> pet.width <- iris[,4,]
```

The matrix `pet.length` contains 50 observations of petal lengths for each of three species of iris: Setosa, Versicolor, and Virginica. The matrix `pet.width` contains 50 observations of petal widths for each of the same three species.

To graphically explore the relationship between petal lengths and petal widths, use `matplot` to display widths versus lengths simultaneously on a single plot:

```
> matplot(pet.length, pet.width, cex = 1.3)
```

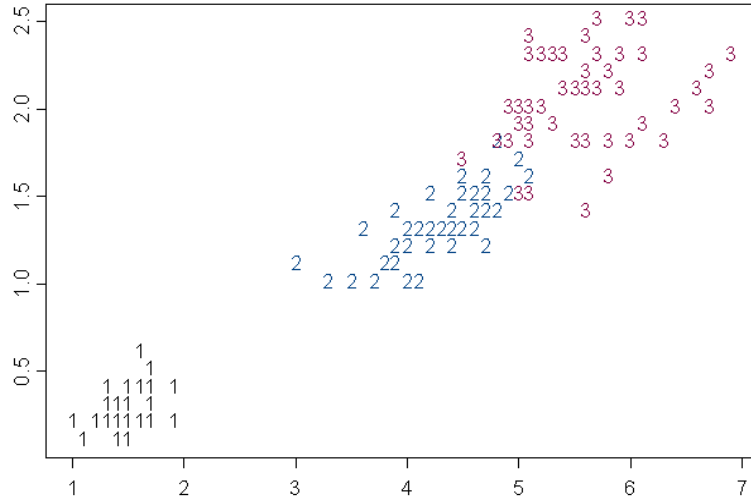


Figure 2.21: Simultaneous plots of petal lengths versus petal widths for three species of iris.

If the matrices you plot with `matplot` do not have the same number of columns, the columns of the smaller matrix are cycled so that every column in the larger matrix is plotted. Thus, if `x` is a vector (i.e., a matrix with a single column), then `matplot(x,y)` plots every column of the matrix `y` against the vector `x`.

Star Plots

A *star plot* displays multivariate data as a set of stars in which each star represents one observation, and each point or *radial* of a star represents a particular variable. The length of each radial is proportional to the data value of the corresponding variable. Thus, both the size and shape of the stars have meaning: size reflects the overall magnitude of the data, and shape reveals the relationships between variables. Comparing two stars gives a quick graphical picture of similarities and differences between two cases; similarly shaped stars indicate similar cases.

For example, to create a star plot of the `longley.x` data, type the following command:

```
> stars(longley.x)
```

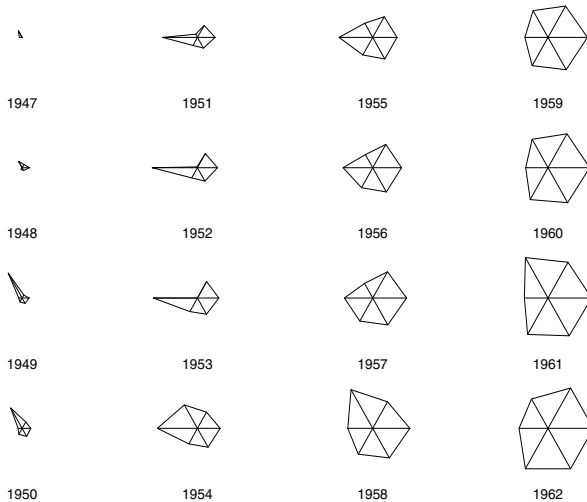


Figure 2.22: A star plot of the `longley.x` data.

Faces

In 1973, Chernoff introduced the idea of using faces to represent multivariate observations. Each variable in a given observation is associated to one feature of the face. Two cases can be compared using a feature-by-feature comparison. You can create Chernoff's faces with the Spotfire S+ `faces` function:

```
> faces(t(cereal.attitude),
+       labels = dimnames(cereal.attitude)[[2]], ncol = 3)
```

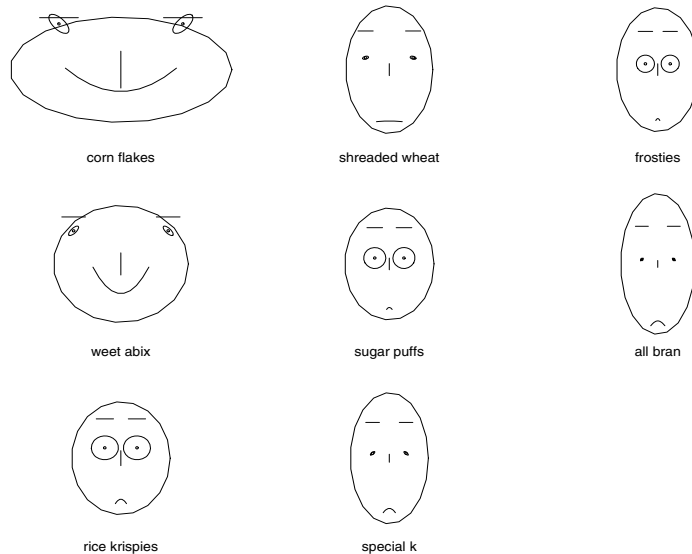


Figure 2.23: A faces plot of the `cereal.attitude` data.

See the `faces` help file and Chernoff (1973) for complete details on interpreting Chernoff faces.

INTERACTIVELY ADDING INFORMATION TO YOUR PLOT

The functions described so far in this chapter create complete plots. Often, however, you want to build on an existing plot in an interactive way. For example, you may want to identify individual points in a plot and label them for future reference. Or you may want to add some text or a legend, or overlay some new data. In this section, we describe some simple techniques for interactively adding information to your plots. More involved techniques for producing customized plots are described in the section Customizing Your Graphics (page 72).

Identifying Plotted Points

While examining a plot, you may notice that some of the plotted points are unusual in some way. To identify the observation numbers of such points, use the `identify` function, which lets you “point and click” with a mouse on the unusual points. For example, consider the plot of y versus x :

```
> set.seed(12)
> x <- runif(20)
> y <- 4*x + rnorm(20)
> x <- c(x,2)
> y <- c(y,2)
> plot(x,y)
```

You immediately notice one point separated from the bulk of the data. Such a data point is called an *outlier*. To identify this point by observation number, use `identify` as follows:

```
> identify(x, y, n=1)
```

After pressing RETURN, you *do not* get a prompt. Instead, Spotfire S+ waits for you to identify points with the mouse. Move the mouse cursor into the graphics window so that it is adjacent to the data point to be identified, and click the left mouse button. The observation number appears next to the point. If you click when the cursor is more than 0.5 inches from the nearest point in the plot, a message appears on your screen to tell you there are no points near the cursor.

After identifying all the points that you requested, Spotfire S+ prints out the observation numbers of the identified points and returns your prompt:

```
> identify(x, y, n=1)
[1] 21
```

If you omit the optional argument *n*, you can identify as many points as you wish. In this case, you must signal Spotfire S+ that you've finished identifying points by taking an appropriate action (i.e., pressing the right mouse button, or pressing both the left and right mouse buttons together, depending on your configuration).

Adding Straight Line Fits to a Scatter Plot

When you create a scatter plot, you may notice a linear association between the *y*-axis variable and the *x*-axis variable. In such cases, you may find it helpful to display a straight line that has been fit to the data. You can use the function `abline(a,b)` to add a straight line with intercept *a* and slope *b* to the plot. The examples below show how to do this for different types of lines.

Least-squares straight line

The best-known approach to fitting a straight line to a scatter plot is the method of least squares. The Spotfire S+ function `lm` fits a linear model using the method of least squares. The `lm` function requires a formula argument, expressing the dependence of the response variable *y* on the predictor variable *x*. See the *Guide to Statistics* for a complete description of formulas and statistical modeling.

To draw a least-squares line on a scatter plot, use `abline` on the results of `lm`. The following Spotfire S+ expressions plot a dotted line for the least-squares fit in a scatter plot of *y* vs. *x*:

```
> plot(x,y)
> abline(lm(y~x), lty = 2)
```

Robust straight line

While fitting a least-squares line to two-dimensional data is probably the most common fitting procedure, the least-squares approach has a fundamental weakness: it lacks robustness, in that it is very sensitive to outliers. A *robust* method is one that is not greatly affected by outliers, providing a good fit to the *bulk* of the data. Spotfire S+ has many functions for computing robust regression lines, including:

- Least trimmed squares regression (LTS), available through the function `ltsreg`
- Least median squares regression (LMS), available through the function `lmsreg`
- Robust MM regression, available through the function `lmRobMM`.

Each of these functions can be used with `abline` to draw a robust straight line on a scatter plot of data. For example, to draw a robust MM line, type the following commands:

```
> plot(x,y)
> abline(lmRobMM(y~x), lty = 2)
```

See the *Guide to Statistics* for a detailed discussion of robust regression techniques.

Adding New Data to the Current Plot

Once you have created a plot, you may want to add additional data to it. For example, you might plot an additional data set with a different line type or plotting character. Or you might add a statistical function such as a smooth curve fit to the data already in the plot. To add data to a plot created by the `plot` function, use one of the two functions `points` or `lines`. These functions are virtually identical to `plot` except that they don't create a new set of axes. The `points` function is used to add data points, while `lines` is used to add lines.

All of the arguments to `plot` that we've discussed so far in this chapter (including `type`, `pch`, and `lty`) work with `points` and `lines`. This means that you can choose line types and plotting characters as you wish. You can even make line-type plots with `points` and `points`-type plots with `lines` if you choose. For example, suppose you plot the built-in data set `co2`, which gives monthly levels of carbon dioxide at the Mauna Loa volcano from January 1959 to December 1990:

```
> plot(co2)
```

By default, `plot` uses the `points` function to plot the data. The `plot` function recognizes that `co2` is a time series data set consisting of monthly measurements, and provides appropriate labels on the horizontal axis. The series `co2` has an obvious seasonal cycle and an increasing trend. It is often useful to smooth such data and display the smoothed version in the same plot. The function `smooth` produces a

smoothed version of a Spotfire S+ time series, and can be used as an argument to `lines`. This adds a plot of the smoothed version of the time series to the existing plot, as shown in the following command:

```
> lines(smooth(co2))
```

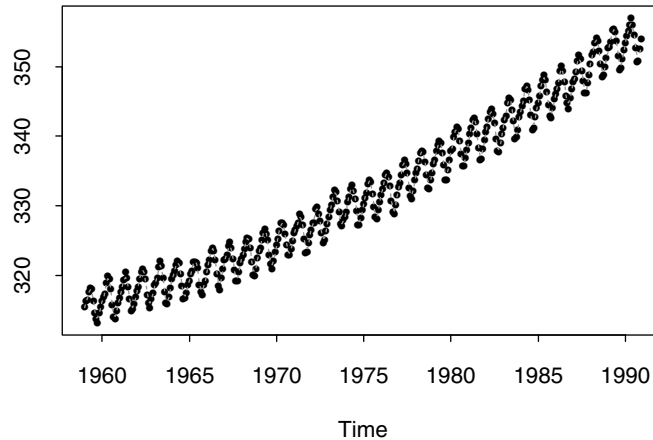


Figure 2.24: *The co2 data with a smoothed line plot.*

If your original plot was created with `matplot`, you can also add new data to it with functions analogous to `points` and `lines`. To add data to a plot created with `matplot`, use `matpoints` or `matlines`. See the corresponding help files for further details.

Warning

If the data you add with `points` or `lines` have a range greater than the axis limits of the original plot, Spotfire S+ does not add all of the data to the plot. Instead, you receive an “out of bounds” warning message, and only the data within the axis limits are plotted. You can avoid this with appropriate use of the optional arguments `xlim` and `ylim` in your call to `plot`.

Adding Text to Your Plot

Suppose you want to add some text to an existing plot. For example, consider the automobile mileage data plot in Figure 2.5. To add the text “Outliers” near the three outlying data points in the upper right corner of the plot, use the `text` function. To use `text`, specify the x and y coordinates at which you want the text to appear, in the same

coordinate system used for the plot itself. More generally, you can specify vectors of x and y coordinates and a vector of text labels. Thus, in the mileage example, type:

```
> plot(car.miles, car.gals)
> text(275, 22, "Outliers")
```

The text “Outliers” is centered at the xy coordinates 275,22. You can guess the coordinate values by “eyeballing” the spot on the plot where you want the text to go. However, this approach to locating text is not very accurate. Instead, you can specify the coordinates of the text exactly using the `locator` function within `text`. The `locator` function allows you to use the mouse cursor to accurately identify the location of any number of points on your plot. When you use `locator`, Spotfire S+ waits for you to position the mouse cursor and click the left mouse button, and then it calculates the coordinates of the selected point. The argument to `locator` specifies the number of times the text is to be positioned. For example, we can apply `text` and `locator` together on the plot of the mileage data as follows:

```
> text(locator(1), "Outliers")
```

Connecting text and data points with straight lines

In the above example, suppose you want to improve the graphical presentation by drawing a straight line from the text “Outliers” to each of the three data points that you regard as outliers. You can add such lines sequentially with the following expression:

```
> locator(n=2, type="l")
```

Spotfire S+ awaits your response. To draw a line, locate the mouse cursor at the desired starting point for the line and click the left button. Move the mouse cursor to the desired ending point for the line and click the left button again. Spotfire S+ draws a straight line between the two points and returns their coordinates at the command prompt. The argument `n=2` tells Spotfire S+ to locate a maximum of two points; to draw additional lines, you must increase the value of `n` appropriately.

Adding legends

Often, you create plots that contain one or more sets of data displayed with different plotting characters or line types. In such cases, you may want to provide a legend that identifies each of the plotting characters

or line types. To do this in Spotfire S+, use the `legend` function. For example, suppose you use the following commands to plot the data shown in Figure 2.25:

```
> plot(smooth(co2), type = "l")  
> points(co2, pch = "x")
```

For clarity, you probably want to add the legend shown in the figure. First, create a vector `leg.names` that contains the character strings "co2" and "smooth of co2". You can then use `legend` as follows:

```
> leg.names <- c("co2", "smooth of co2")  
> legend(locator(1), leg.names, pch = "x ", lty = c(0,1))
```

Spotfire S+ waits for you to respond. Move the mouse cursor to the location on the plot where you want to place the *upper left corner* of the legend box, then click the left mouse button.

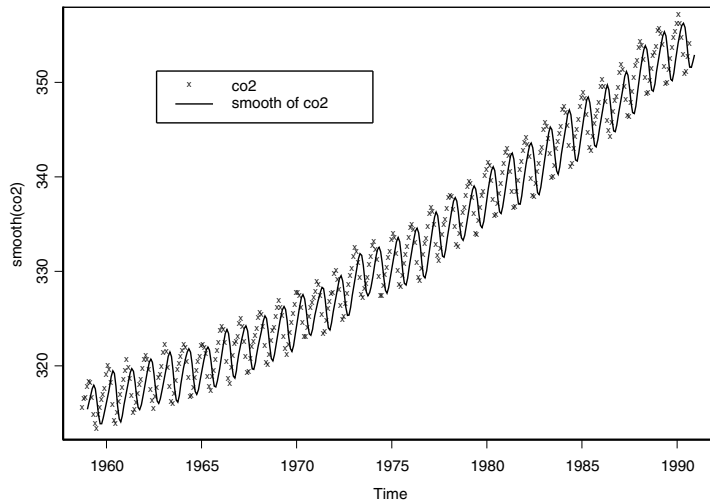


Figure 2.25: *A plot of the co2 data with a legend added.*

CUSTOMIZING YOUR GRAPHICS

For most exploratory data analysis, the complete graphics created by Spotfire S+ should serve your needs well, with automatically generated axes, tick marks, and axis labels. Most of the graphics described in the previous sections are created with one-step functions such as `plot` and `hist`. These one-step functions are called *high-level* graphics functions. If you are preparing graphics for publication or a presentation, you may need more control over the graphics that Spotfire S+ produces.

This section and the remaining sections in this chapter describe how to customize and fine-tune your Spotfire S+ graphics with *low-level* graphics functions and *graphics parameters*. Low-level graphics functions do not generate a complete graphic, but rather one specific part of it. Graphics parameters control the details of the plots that are produced by the graphics functions, including where the graphics appear on the device. In Windows, the customization features described here refer to traditional command-line graphics; you may prefer the commands described in the Chapter 6, Windows Editable Graphics Commands as your customization method.

Many of the remaining examples in this chapter use the following data:

```
> set.seed(12)
> x <- runif(12)
> y <- rnorm(12)
```

We also use the following data from the built-in data set `auto.stats`:

```
> price <- auto.stats[, "Price"]
> mileage <- auto.stats[, 2]
```

Low-level Graphics Functions and Parameters

The section Frequently Used Plotting Options (page 33) introduces several *low-level* graphics functions, including `points`, which adds a scatter of points to an existing plot, and `abline`, which adds a specified line to an existing plot. Low-level graphics functions, unlike high-level functions, do not automatically generate a new coordinate system. Thus, you can use several low-level graphics functions in succession to create a single finished plot. Note that some functions,

such as the `image` and `contour` commands described in the section *Visualizing Three-Dimensional Data* (page 57), can be used as either high- or low-level graphics functions.

Graphics parameters add to the flexibility of graphics by controlling virtually every detail in a page of graphics. There are about sixty parameters in total, which fall into four classes:

- *High-level* graphics parameters can be used only as arguments to high-level graphics functions. An example is `xlim`, which gives the approximate limits for the x axis.
- *Layout* graphics parameters can be set only with the `par` function. These parameters typically affect quantities that concern the graphics page as a whole. The `mfrac` parameter, which determines how many rows and columns of plots are placed on a single page, is one example.
- *General* graphics parameters (e.g., `col`, `cex`) may be set with either a call to a graphics function or with the `par` function. When used in a graphics function, the change is valid only for that function call. If you set a parameter with `par`, the change lasts until you change it again.
- *Information* parameters provide information about the state of the device, but may not be changed directly by the user. An example is `din`, which stores the size of the current device in inches. See the `par` help file for descriptions of the information parameters.

Graphics parameters are initialized whenever a graphics device is started. A change to a parameter via the `par` function applies only to the current device. You can write your own `Device.Default` function to have one or more parameters set automatically when you start a graphics device; see the `Device.Default` help file for more details.

The arguments to `title` (`main`, `sub`, `xlab`, `ylab`, `axes`, ...) are not technically graphics parameters, though they are quite similar to them. They are accepted as arguments by several graphics functions as well as the `title` function.

Table 2.10 (page 108) summarizes all of the Spotfire S+ graphics parameters.

Warning

Some graphics functions do not recognize certain high-level or general graphics parameters. The help files for these functions describe which graphics parameters the functions accept.

Setting and Viewing Graphics Parameters

There are two ways to set graphics parameters:

1. Use the *name=value* form, either within a graphics function call or with the `par` function. For example:

```
> par(mfrow = c(2,1))
> plot(x, y, pch = 17)
> plot(price, mileage, log = "y")
```

Note that you can set several graphics parameters simultaneously in a single call to `par`.

2. Supply a list to the `par` function. The names of the list components are the names of the graphics parameters you want to set. For example:

```
> my.list <- list(mfrow = c(2,1))
> par(my.list)
```

Note

For more information about setting parameters with the `par` function, see the sections Additional Graphics Arguments (page 17), Vectorized Graphics Parameters (page 21), and the `par` help file.

Restoring Original Values

When you change graphics parameters with `par`, it returns a list containing the *original* values of the graphics parameters that you changed. This list does not print out on your screen. Instead, you must assign the result to a variable name if you want to see it:

```
> par.orig <- par(mfrow = c(2,1))
> par.orig
$mfrow:
[1] 1 1
```

You can use the list returned by `par` to restore parameters after you have changed them:

```
> par.orig <- par(mfrow = c(2,1))
> # Now make some plots
> par(par.orig)
```

When setting multiple parameters with `par`, check for possible interactions between parameters. Such interactions are indicated in Table 2.3 and in the `par` help file. In a single call to `par`, general graphics parameters are set first, then layout graphics parameters. If a layout graphics parameter affects the value of a general graphics parameter, what you specify for the general graphics parameter may be overridden. For example, changing `mfrow` automatically resets `cex`; see the section Controlling Multiple Plots (page 95) for more details.

If you type

```
> par(mfrow = c(2,1), cex = 0.75)
```

Spotfire S+ first sets `cex=0.75` because `cex` is a general graphics parameter, and then sets `mfrow=c(2,1)` because `mfrow` is a layout graphics parameter. However, setting `mfrow=c(2,1)` automatically sets `cex` back to 1. To set both `mfrow` and `cex`, you need to call `par` twice:

```
> par(mfrow = c(2,1))
> par(cex = 0.75)
```

Table 2.3: *Interactions between graphics parameters.*

Parameters	Interaction
<code>cex</code> , <code>mex</code> , <code>mfrow</code> , <code>mfcpl</code>	If <code>mfrow</code> or <code>mfcpl</code> specify a layout with more than two rows or columns, <code>cex</code> and <code>mex</code> are set to 0.5. Otherwise, <code>cex</code> and <code>mex</code> are both set to 1.
<code>crt</code> , <code>srt</code>	When <code>srt</code> is set, <code>crt</code> is set to the same value unless <code>crt</code> appears later in the command than <code>srt</code> .

You can also use the `par` function to view the current setting of any graphics parameter. To view the current values of parameters, call `par` with a vector of character strings naming the parameters. For example:

```
> par("usr")
> par(c("mfrow","cex"))
```

To get a list of all of the parameter values, call `par` with no arguments:

```
> par()
```

During an extended Spotfire S+ session, you may make repeated calls to `par` to change graphics parameters. Sometimes, you may forget what you have changed and you may want to restore the device to its original defaults. It is often a good idea to save the original values of the graphics parameters as soon as you start a device. You can then call `par` to restore the device to its original state. The commands below show one approach to this.

```
> par.orig.wg <- par()
> par(mfrow = c(3,1), col = 4, lty = 2)
> # create some plots
> # several more calls to par
> par(par.orig.wg)
```

Warning

When a device is first started, the graphics parameter `new` is set equal to `TRUE` before any plots are produced. In this case, a call to a high-level graphics function does not clear the device before displaying a new plot; see the section [Overlaying Figures](#) (page 97) for more details. Thus, if you follow the above commands to restore all graphics parameters to their original state, you need to call `frame` before issuing the next plotting command.

Separate sets of graphics parameters are maintained for each active graphics device. When you change graphics parameters with the `par` function, you change the values only for the current graphics device. For example, if you open both a graphsheet (on Windows) or motif (on UNIX) and a postscript graphics device and the postscript device is the current one, then changing graphics parameters via `par` affects only the graphics parameters for the postscript device.

The commands below illustrate this using the `dev.list`, `dev.cur`, and `dev.set` functions on Windows.

```
# Open two graphics devices and see which one is current.
> graphsheet()
> postscript()

> dev.list()
  graphsheet postscript
           2           3

> dev.cur()
postscript
          3

# Change a graphics parameter, which affects
# only the postscript device.
> par(mfrow = c(2,2))
> par("mfrow")
[1] 2 2

# Change the active graphics device and
# check the value of the mfrow parameter.
> dev.set()
  graphsheet
           2

> par("mfrow")
[1] 1 1
```

The commands below illustrate this using the `dev.list`, `dev.cur`, and `dev.set` functions on UNIX.

```
# Open two graphics devices and see which one is current.
> motif()
> postscript()

> dev.list()
  motif postscript
           2           3

> dev.cur()
postscript
          3
```

Chapter 2 *Traditional Graphics*

```
# Change a graphics parameter, which affects
# only the postscript device.
> par(mfrow = c(2,2))
> par("mfrow")
[1] 2 2

# Change the active graphics device and
# check the value of the mfrow parameter.
> dev.set()
      motif
      2
> par("mfrow")
[1] 1 1
```


CONTROLLING GRAPHICS REGIONS

The location and size of a figure are determined by parameters that control *graphics regions*. The surface of any graphics device can be divided into two regions: the *outer margin* and the *figure region*. The figure region contains one or more figures, each of which is composed of a *plot region* surrounded by a *margin*. By default, a device is initialized with one figure and the outer margin has zero area; that is, typically there is just a plot region surrounded by a margin.

The plot region is where the data is displayed. In a typical plot, the axis line is drawn on the boundary between the plot area and the margin. Each margin, whether the outer margin or a figure margin, is divided into four parts as shown in Figure 2.26: bottom (side 1), left (side 2), top (side 3), and right (side 4).

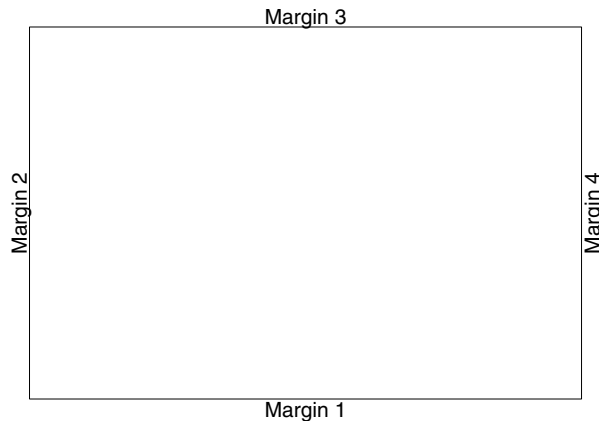


Figure 2.26: *The four sides of a margin.*

You can change the size of any of the graphics regions. Changing one area causes Spotfire S+ to automatically resize the regions within and surrounding the one you have changed. For example, when you specify the size of a figure, the margin size is subtracted from the figure size to obtain the size of the plot area. Spotfire S+ does not allow a plot with a margin to take more room than the figure.

Most often, you change the size of graphics regions with the `mfrow` or `mfcol` layout parameters. When you specify the number of rows and columns in your page of graphics, Spotfire S+ automatically

determines the appropriate figure size. To control region size explicitly, you should work your way inward by specifying first the outer margins, and then the figure margins.

The Outer Margin

You usually specify an outer margin only when creating multiple figures per page. You can use the outer margin to hold a title for an entire page of plots, or to label different pages consistently when some pages have multiple plots and others have a single plot.

You must specify a size for the outer margin if you want one; the default size is 0. To specify the size of the outer margin, use any one of three equivalent layout parameters: `oma`, `omi`, or `omd`. The most useful of these is `oma`, which is a numeric vector of length four (one element for each side of the margin). The values given to `oma` are expressed in `mex`, the size of the font for one line of text in the margins. If you specify the outer margin with `oma`, the four values correspond to the number of lines of text that will fit in each side of the margin. For example, to leave room for a title at the top of a page of plots, we could set the outer margin as follows:

```
> par(oma = c(0,0,5,0))
```

You can then use `mtext` as follows to add a title:

```
> mtext("A Title in the Outer Margin",  
+       side = 3, outer = T, cex = 2)  
> box()
```

The result is shown in Figure 2.27. Setting the parameter `oma` automatically changes both `omi` (the outer margin in inches) and `omd` (the outer margin as a fraction of the device surface). In general, the `oma` parameter is more useful than `omi` because it can be used to specify *relative* margin sizes. Conversely, the `omi` parameter measures the size of each side of the margin in inches, and is thus useful for specifying *absolute* margin sizes. See the `par` help file for more information on `omi` and `omd`.

A Title in the Outer Margin



Figure 2.27: *A plot with an outer margin to hold a main title.*

Warning

If you set `oma` to something other than the default value `c(0,0,0,0)` and then later reset *all* of the graphics parameters in a call to `par`, you will see the warning message:

Warning messages:

Graphics error: Figure specified in inches too large (in `zzfigz`) in:...

This message can be safely ignored.

Figure Margins To specify the size of the figure margins, use one of two equivalent graphics layout parameters: `mar` or `mai`. The `mar` parameter is specified as a numeric vector of length four (one element for each side of the margin) with values expressed in `mex`. It is generally more useful than `mai` because it can be used to specify *relative* margin sizes. Conversely, the `mai` parameter measures the size of each side of the margin in inches, and is thus useful for specifying *absolute* margin sizes. For example, if `mex` is the default value of 1 and `mar` equals `c(5,5,5,5)`, there is room for five lines of default-font text (`cex=1`) in each margin. If `mex` is 2 and `mar` is `c(5,5,5,5)`, there is room for 10 lines of default-font text in each margin.

The `mex` parameter specifies the size of font that is to be used to measure the margins. When you change `mex`, Spotfire S+ automatically resets some margin parameters to decrease the size of

the figure margins, but without changing the size of the outer margin. Table 2.4 shows the effects on the various margin parameters when `mex` is changed from 1 to 2.

Table 2.4: *Effects of changing the `mex` parameter.*

Parameter	mex=1	mex=2
<code>mar</code>	5.1 4.1 4.1 2.1	5.1 4.1 4.1 2.1
<code>mai</code>	0.714 0.574 0.574 0.294	1.428 1.148 1.148 0.588
<code>oma</code>	0 0 5 0	0.0 0.0 2.5 0.0
<code>omi</code>	0.000 0.000 0.699 0.000	0.000 0.000 0.699 0.000

From the table, we see that an increase in `mex` leaves `mar` and `omi` unchanged, while `mai` is increased and `oma` is decreased. When you shrink margins with `mar`, be sure to check the `mfp` parameter, which determines where axis and tick labels are placed. If the margins don't provide room for those labels, the labels are not printed and you receive a warning from Spotfire S+.

The Plot Region

To determine the shape of a plot, use the `pty`, or “plot type” layout graphics parameter. The `pty` parameter has two possible values: “m” for maximal and “s” for square. By default, `pty=“m”` and a plot fills its entire allotted space. Another way to control the shape of a plot is with the `pin` parameter, which gives the width and height of the plot in inches.

CONTROLLING TEXT AND SYMBOLS

The section Interactively Adding Information to Your Plot (page 66) describes how to add text and legends to existing plots. This section describes how to control the size of text and plotting symbols, the placement and orientation of text within the plot region, and the width of plotted lines.

Text and Symbol Size

The size of text and most plotting symbols is controlled by the general “character expansion” parameter `cex`. The *expansion* term refers to expansion with respect to the default font of the graphics device. By default, `cex` is set to 1, so graphics text and symbols appear in the default font size. When `cex=2`, text appears at twice the default font size. Some devices, however, have only a few fonts available, so that all values of `cex` in a certain range produce the same font.

Note

The `plot` function, and functions that add individual points to a plot, can accept a vector of values for the `pch`, `cex`, and `col` parameters. For example:

```
plot(halibut$biomass, cex = rep(1:3,each=20))
```

For more information, see the section Vectorized Graphics Parameters (page 21).

Many graphics functions and parameters use or modify `cex`. For example, main titles are written with a character expansion of 1.5 times the current `cex`. The `mfrow` parameter sets `cex` to 1 for small numbers of plots (fewer than three per row or column), but sets it to 0.5 for larger numbers of plots.

The `cex` parameter controls the size of both text and plotting symbols. Figure 2.28 shows how symbols of different sizes can be used to highlight groups of data. The figure is produced with the following expressions:

```
> plot(x, y, pch = 16)
> points(
+   x[x-y > 2*median(x-y)], y[x-y > 2*median(x-y)],
+   pch = 16, cex = 2)
```

```
> points(x[x-y < median(x-y)], y[x-y < median(x-y)],  
+       pch = 18, cex = 2)
```

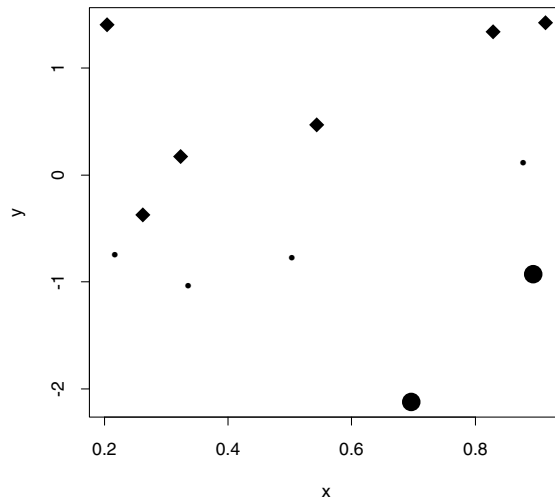


Figure 2.28: *Plotting symbols of different sizes.*

A parameter equivalent to `cex` is `csi`, which gives the height (interline space) of text with the current `cex` measured in inches. Changing one of `cex` or `csi` automatically changes the other. The `csi` parameter is useful when creating the same graphics on different devices, since the absolute size of graphics is device dependent.

Text Placement

When you add text to the plot region in a figure, you specify its coordinates in terms of the plotted data. In essence, Spotfire S+ treats the added text as a data point. If axes have been drawn and labeled, you can read the desired coordinates off the plot. If not, you can obtain the coordinates by interpolating from the values in the layout parameter `usr`.

For example, Figure 2.28 has an x axis with values from 0 to 1 and a y axis with values running from approximately -2.5 to 1. To add the text “Different size symbols” to the plot, we could specify any point within the grid determined by these x and y limits:

```
> text(0.4, 0.7, "Different size symbols")
```

By default, the text is centered at the specified point. You can left- or right-justify the text by using the general graphics parameter `adj`. The `adj` parameter determines the fraction of the text string that appears to the left of the specified coordinate. The default value is 0.5, which places approximately half of the text string to the left of the coordinate. Set `adj=0` to left-justify, and `adj=1` to right-justify.

If no axes have been drawn on your plot and you cannot determine coordinates by simply looking at your graphic, you can interpolate from the values in the layout parameter `usr`. The `usr` parameter gives the minimum and maximum of the x and y coordinates in the plot. Typing `par("usr")` returns the extremes of the x and y data, from which you can guess the coordinates of the desired location of your text.

It is also possible to use the `locator` function with `text` to interactively choose a location in your plot without explicitly knowing the coordinates. For examples of this technique, see the section *Adding Text to Your Plot* (page 69).

Text Orientation

Two graphics parameters control the orientation of text in the plot region, the figure, and the outer margins: `crt` (“character rotation”) and `srt` (“string rotation”). Figure 2.29 shows the result of typing the following commands after starting a postscript device:

```
> plot(1:10, type = "n")
> text(2, 2, "srt=0, crt=0", srt = 0, crt = 0)
> text(4, 4, "srt=0, crt=90", srt = 0, crt = 90)
> text(6, 6, "srt=90, crt=0", srt = 90, crt = 0)
> text(8, 8, "srt=90, crt=90", srt = 90, crt = 90)
```

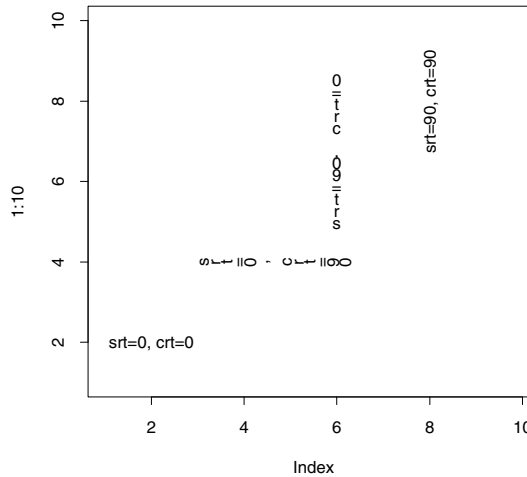


Figure 2.29: Character and string rotation using the `crt` and `srt` parameters.

The postscript device is the only graphics device that uses both the `crt` and `srt` graphics parameters. All other graphics devices ignore `crt`, so you can rotate only the whole string with `srt`.

Warning

If you use both `crt` and `srt` in a plotting command while running the postscript device, you must supply `crt` *after* `srt`; otherwise, it will be ignored.

Text in Figure Margins

To add text in figure margins, use the `mtext` marginal text function. You can specify the side of the margin where you want text with the `side` argument, which is a number from 1 to 4. The default value is 3, which places text at the top of the plot. The `line` argument to `mtext` gives the distance in `mex` between the text and the plot. For example, Figure 2.30 shows the placement of the following marginal text:

```
> par(mar = c(5,5,5,5) + 0.1, pty = "s")
> plot(x, y, type = "n", axes = F, xlab = "", ylab = "")
> box()
> mtext("Some text", line = 0)
> mtext("Some more text", side = 2, cex = 1, line = 2)
> mtext("Still more text", side = 4, cex = 0.5, line = 3)
```

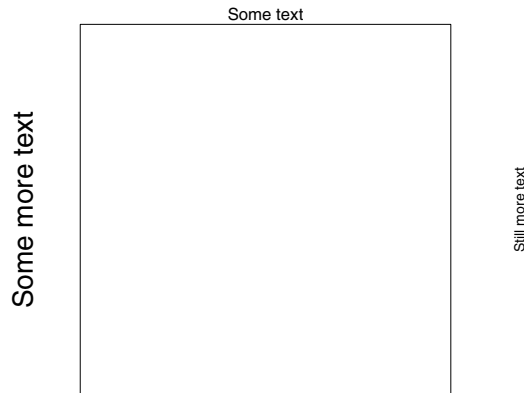



Figure 2.30: *Placing text in figure margins using the `mtext` function.*

Text is not placed in the margin if there is no room for it. This usually happens only when the margin sizes or `cex` have been reset, or when axis labels are extremely long. For example, suppose `mex=1` and you reset the figure margins with `mar=c(1,1,1,1)` to allow precisely one line of text in each margin. If you try to write text in the margins with the parameter value `cex=2`, it will not fit because the text is twice as high as the specified margin line.

To specify the position of the text *along* the margin, you can use the `at` argument to the `mtext` function. The value of the `at` argument is in units of the x or y coordinates, depending on whether you are placing text on the top or bottom margin (sides 1 and 3), or the left or right margin (sides 2 and 4). As described in section Text Placement (page 84), if you can't determine the appropriate value of the `at` argument, you can interpolate from the value of `usr` graphics parameter. For example, the following command places text in the lower left corner of the margin in Figure 2.30:

```
> par("usr")
[1] 0.1758803 0.9420847 -2.2629721 1.5655365

> mtext("A comment", line = 3, side = 1, at = 0.3)
```

By default, `mtext` centers text along the margin, or at the `at` coordinate if one is supplied. You can also use the `adj` parameter to place text along the margin. The default setting is `adj=0.5`, which results in centered text. Set `adj=0` to place the text flush with the left

side of the margin (or with the `at` coordinate), and set `adj=1` to place the text flush right. Values of `adj` between 0 and 1 place the text so that the specified fraction of the string is placed before the given coordinate.

Note

The `adj` parameter is generally more useful than `usr` when writing in the outer margin of multiple figures, because the value of `usr` is the coordinates from the most recent plot created in the figure region.

By default, `mtext` rotates text to be parallel to the axis. To control the orientation of text in the margins, use the `srt` parameter along with the `at` argument. For example, the following command displays upside-down text in the top figure margin:

```
> mtext("Title with srt=180", line=2, at=0.5, srt=180)
```

Warning

If you supply `mtext` with the `srt` argument, you must also specify the `at` argument. Otherwise, `srt` is ignored.

Plotting Symbols in Margins

In general, Spotfire S+ clips plotting symbols so that they do not appear in the margins. You can allow plotting in the margin by setting the `xpd` graphics parameter to `TRUE`. This parameter value expands the allowable plotting area and results in unclipped symbols.

Line Width

The width of lines, both within a plot and on the axes, is controlled by the general graphics parameter `lwd`. The default value of `lwd` is 1; larger values produce wider lines and smaller values produce narrower lines. Note that line width is device dependent, and some graphics devices can produce only one width.

CONTROLLING AXES

The high-level graphics commands described in the section *Getting Started with Simple Plots* (page 30) create complete graphics, including labeled axes. However, you may need to create graphics with axes different from those provided by Spotfire S+. For example, you may need to specify a different choice of axes, different tick marks, or different plotting characteristics. This section describes how to control the look of the axes in your plots.

Enabling and Disabling Axes

Whether axes appear on a plot is determined by the high-level graphics parameter `axes`, which takes a logical value. If `axes=FALSE`, no axes are drawn on the plot. If axes are not drawn on the original plot, they can be added afterward with one or more calls to the `axis` function.

You can use `plot` with `axes=F`, together with the `axis` function, to create plots of mathematical functions on a standard Cartesian coordinate system. For example, the following simple function plots a set of points from the domain of a function against the set's image on a Cartesian grid:

```
> mathplot <- function(domain,image) {
+   plot(domain, image, type = "l", axes = F)
+   axis(1, pos = 0)
+   axis(2, pos = 0)
+ }
```

Tick Marks and Axis Labels

To control the length of tick marks in a plot, use the `tck` general graphics parameter. This parameter is a single number that is interpreted as a fraction of a plot dimension. If `tck` is less than 0.5, the tick marks on each axis have the same length; this length is the fraction `tck` of the smaller of the width and height of the plot area. Otherwise, the length of the tick marks on each axis are a fraction of the corresponding plot dimension. The default value is `tck=-0.02`, resulting in tick marks on each axis that have equal length and are pointing out from the plot. Use `tck=1` to draw grid lines in the plot region.

To familiarize yourself with the `tck` parameter, try the following commands:

```
> par(mfrow = c(2,2))
> plot(x, y, main = "tck = -0.02")
> plot(x, y, main = "tck = 0.05", tck = 0.05)
> plot(x, y, main = "tck = 1", tck = 1)
```

It is possible to have tick marks of different lengths and styles on each axis. The following code first draws a plot with no axes, then adds each axis individually with different values of the `tck` and `lty` parameters:

```
> plot(x, y, axes = F, main = "Different tick marks")
> axis(1)
> axis(2, tck = 1, lty = 2)
> box()
```

Note

For information about using separate color arguments to control the color of the individual elements of a plot (`col.axis`, `col.lab`, `col.main`, `col.sub`, `fg`, and `bg`) see the section Additional Graphics Arguments (page 17).

To control the number of tick marks on an axis, you can set the `lab` parameter. The `lab` parameter is an integer vector of length three that gives the approximate number of tick marks on the x axis, the approximate number of tick marks on the y axis, and the number of characters for tick labels. The numbers are only approximate because Spotfire S+ tries to use rounded numbers for tick labels. It may take some experimentation with the `lab` parameter to obtain the axis that you want.

To control the format of tick labels in exponential notation, use the `exp` graphics parameter. Table 2.5 lists the effects of setting `exp=0`, `exp=1`, and `exp=2`.

Table 2.5: *Controlling the format of tick labels with the `exp` graphics parameter.*

Setting	Effect
<code>exp=0</code>	Exponential tick labels are printed on two lines, so that <code>2e6</code> is printed with <code>2.0</code> on one line and <code>e6</code> on the next.
<code>exp=1</code>	Exponential tick labels are printed on a single line, in the form <code>2.0e6</code> .
<code>exp=2</code>	The default value. Exponential tick labels are printed on a single line, in the form <code>2*10^6</code> .

Uses of the `lab` and `exp` parameters are illustrated with the code below.

```
> par(mfrow = c(2,2))
> plot(price, mileage, main = "lab = c(5,5,7)")
> plot(price, mileage,
+      main = "lab = c(10,3,7)", lab = c(10,3,7))
> plot(price, mileage,
+      main = "lab = c(5,5,4), exp = 2", lab = c(5,5,4))
> plot(price, mileage,
+      main = "lab = c(5,5,4), exp = 1",
+      lab = c(5,5,4), exp = 1)
```

To control the orientation of the axis labels, use the `las` graphics parameter. You can choose between labels that are written parallel to the axes (`las=0`), horizontally (`las=1`), or perpendicular to the axes (`las=2`). By default, `las=0`. To see the effect of this parameter, try the following commands:

```
> par(mfrow = c(2,2))
> plot(x, y, main = "Parallel, las=0", las = 0)
> plot(x, y, main = "Horizontal, las=1", las = 1)
> plot(x, y, main = "Perpendicular, las=2", las = 2)
```

```
> plot(x, y, main = "Customized", axes = F)
> axis(2)
> axis(1, at = c(0.2, 0.4, 0.6, 0.8),
+   labels = c("2/10", "4/10", "6/10", "8/10"))
> box()
```

The `box` function ensures that a complete rectangle is drawn around the plotted points; see the section *Axis Boxes* (page 94) for more details.

The `xaxt` and `yaxt` graphics parameters also control axis plotting. If one of these parameters is equal to "n", the tick marks for the corresponding axis are not drawn. For example, you could create the last panel produced by the code above with the following commands:

```
> plot(x, y, main = "Customized", xaxt = "n")
> axis(1, at = c(0.2, 0.4, 0.6, 0.8),
+   labels = c("2/10", "4/10", "6/10", "8/10"))
```

To set the distance from the plot to an axis title, use the `mgp` general graphics parameter. The `mgp` parameter is a numeric vector with three elements in units of `mex`: the first element gives the location of the axis title, the second element gives the location of the tick labels, and the third gives the location of the axis line. The default value is `c(3, 1, 0)`. You can use `mgp` to control how much space the axes consume. For example, if you have small margins, you might create a plot with:

```
> plot(x, y, tck = 0.02, mgp = c(2, 0.1, 0))
```

This draws the tick marks inside the plot and brings the labels closer to the axis line.

Axis Style

The `xaxs` and `yaxs` parameters determine the style of the axes in a plot. The available styles are listed in Table 2.6.

Table 2.6: *Axis styles governed by the `xaxs` and `yaxs` graphics parameters.*

Setting	Style
"r"	The default axis style, also referred to as <i>rational axes</i> . This setting extends the range of the data by 4% and then labels internally. An <i>internally labeled</i> axis has labels that are inside the range of the data.
"i"	Labels internally without expanding the range of the data. Thus, there is at least one data point on each boundary of an "i" style axis (if <code>xlim</code> and <code>ylim</code> are not used).
"e"	Labels externally and expands the range of the data by half a character if necessary, so that no point is precisely on a boundary. An <i>externally labeled</i> axis includes a “pretty” value beyond the range of the data. The "e" style axis is also referred to as an <i>extended axis</i> .
"s"	<i>Standard axes</i> are similar to extended axes but do not expand the range of the data. A plot with standard axes is exactly the same as a plot with extended axes for some data sets, but for others the extended axes contain a slightly wider range.
"d"	<i>Direct axes</i> retain the axes from the previous plot. For example, you can create several plots that have precisely the same <i>x</i> or <i>y</i> axis by setting <code>xaxs="d"</code> or <code>yaxs="d"</code> , respectively. You can include the parameter settings as arguments to the second and subsequent plotting commands, or you can set them with <code>par</code> . If you define direct axes with <code>par</code> , you need to remember to release the axes after you are finished.

Axis styles can be illustrated with the following expressions:

```
> par(mfrow = c(2,2))
> plot(x, y, main = "Rational axes")
> plot(x, y, main = "Internal axes", xaxs = "i", yaxs = "i")
> plot(x, y, main = "Extended axes", xaxs = "e", yaxs = "e")
> plot(x, y, main = "Standard axes", xaxs = "s", yaxs = "s")
```

Axis Boxes

You control boxes around the plot region using the `bty` (“box type”) graphics parameter. This parameter specifies the type of box to be drawn around a plot. The available box types are listed in Table 2.7.

Table 2.7: *Specifying the type of box around a plot, using the `bty` parameter.*

Setting	Effect
"n"	No box is drawn around the plot, although the x and y axes are still drawn.
"o"	The default box type. This setting draws a four-sided box around the plot. The box resembles an uppercase “O,” hence the option name.
"c"	Draws a three-sided box around the plot in the shape of an uppercase “C.”
"l"	Draws a two-sided box around the plot in the shape of an uppercase “L.”
"7"	Draws a two-sided box around the plot in the shape of a square numeral “7.”

The `box` function draws a box of given thickness around the plot area. The shape of the box is determined by the `bty` parameter. You can use `box` to draw full boxes on plots with customized axes, as the commands below illustrate.

```
> par(mfrow = c(2,2))
> plot(x, y, main = "O Box")
> plot(x, y, main = "C Box", bty = "c")
> plot(x, y, main = "L Box", bty = "l")
> plot(x, y, main = "Heavy Box")
> box(20)
```


CONTROLLING MULTIPLE PLOTS

Multiple Figures on One Page

As we have seen earlier in this chapter, multiple figures can be created using `par` and `mfrow`. For example, to set a plot layout of three rows by two columns, use the following command:

```
> par(mfrow = c(3,2))
```

In this section, we describe how to control multiple plots in more detail. In our examples, we use the following data introduced at the beginning of this chapter:

```
> x <- seq(from = 0, to = 20, by = 0.1)
> y <- exp(-x/10) * cos(2*x)
```

When you specify one of the layout parameters `mfrow` or `mfcol`, Spotfire S+ automatically changes several other graphics parameters. The interactions are listed in Table 2.8. To override the values of `mex` and `cex` chosen by `mfrow` and `mfcol`, you must issue separate calls to `par`:

```
> par(mfrow = c(2,2))
> par(mex = 0.6, cex = 0.6)
```

Table 2.8: *Changes in graphics parameters induced by `mfrow` and `mfcol`.*

Parameter	Effects
<code>fty</code>	Is set to "c" by <code>mfcol</code> and to "r" by <code>mfrow</code> . This parameter tells Spotfire S+ whether to place plots along rows or columns in the figure.
<code>mfg</code>	Contains the row and column of the current plot, and the number of rows and columns in the current array of figures.
<code>cex</code> <code>mex</code>	If either the number of rows or the number of columns in the figure is greater than 2, then both <code>cex</code> and <code>mex</code> are set to 0.5.

The `mfrow` and `mfcol` layout parameters automatically create multiple-figure layouts in which all figures are the same size. Instead, you can create multiple-figure layouts in which the figures are different sizes by using the `fig` layout parameter. The `fig` graphics parameter gives the coordinates of the corners of the current figure as fractions of the device surface. An example is shown in Figure 2.31, in which the first plot uses the top third of the device, the second plot uses the left half of the bottom two-thirds of the device, and the last plot uses the right half of the bottom two-thirds. The example begins with the `frame` function, which tells the graphics device to begin a new figure. The commands below reproduce Figure 2.31.

```
> frame()
> par(fig = c(0, 1, 0.66, 1), mar = c(5,4,2,2) + 0.1)
> plot(x)
> par(fig = c(0, 0.5, 0, 0.66))
> plot(x,y)
> par(fig = c(0.5, 1, 0, 0.66))
> plot(y, yaxs = "d")
> par(fig = c(0,1,0,1))
```

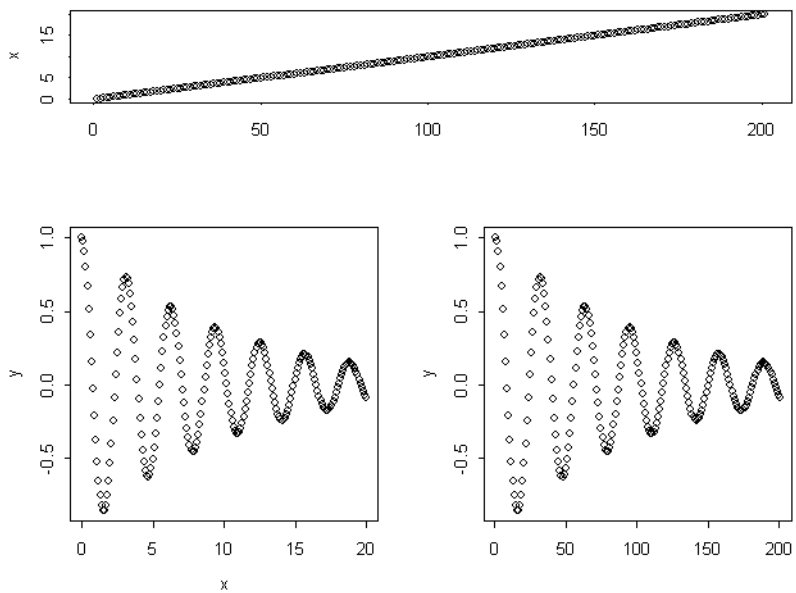


Figure 2.31: Controlling the layout of multiple plots using the `fig` graphics parameter.

Once you create one figure with `fig`, you must use it to specify the layout of the entire page of plots. When you complete your custom plot, reset `fig` to `c(0,1,0,1)`.

Hint

If you want to issue a high-level plotting command in a screen that already has a plot in it, but you don't want the plots in the other screens to disappear, use the `erase.screen` function before calling the high-level plotting command.

Pausing Between Multiple Figures

In screen devices such as `graphsheet` (Windows) or `java.graph`, Spotfire S+ sends a multipage display to different pages of the same window. If a multipage display is sent to a screen device, the default behavior draws each page in order without pausing between pages. You can force the device to prompt you before drawing each page by typing

```
> par(ask=TRUE)
```

before issuing your graphics commands.

The `ask` parameter also forces Spotfire S+ to ask your permission before erasing the graphics on the current device. For example, consider the following plotting commands:

```
> plot(x)
> plot(y)
```

Normally, the second call to `plot` overwrites the first graph on the current device. You can force Spotfire S+ to prompt you before erasing the first graph by calling `par(ask=TRUE)`:

```
> par(ask=TRUE)
> plot(x)
> plot(y)
```

Like all graphics parameters, the `ask` setting remains until the current device is closed.

Overlaying Figures

It is often desirable to include more than one data set on the same plot. As we have seen in this chapter, simple additions can be made with the `lines` and `points` functions. In addition, the `matplot` function plots a number of columns of data at once. These

approaches all assume, however, that the data are all on the same scale. In this section, we discuss several ways of overlaying plots when the data are not necessarily on the same scale.

There are three general ways to overlay figures in Spotfire S+:

1. Call a high-level plotting function, then call one of the high-level plotting functions that can be used as a low-level plotting function by specifying the argument `add=T`.
2. Call a high-level plotting function, set the graphics parameter `new=TRUE`, then call another high-level plotting function.
3. Use the `subplot` function.

We discuss each of these methods below.

High-Level Functions That Can Act as Low-Level Functions

There are currently four plotting functions that can act as either high-level or low-level graphics functions: `usa`, `symbols`, `image`, and `contour`. By default, these functions act like high-level plotting functions. To make them act like low-level plotting functions instead, set the optional argument `add=TRUE`. For example, you can display a map of the northeastern United States with a call to `usa`, then overlay a contour plot of ozone concentrations with a call to `contour` as follows:

```
> usa(xlim = range(ozone.xy$x), ylim = range(ozone.xy$y),  
+     lty = 2, col = 2)  
  
> contour(interp(ozone.xy$x, ozone.xy$y, ozone.median),  
+         add = T)  
  
> title("Median Ozone Concentrations in the North East")
```

Setting the Argument `new=TRUE`

Another way to overlay figures is to reset the `new` graphics parameter. When a graphics device is initialized, the graphics parameter `new` is set to `TRUE` to indicate that it is a new graphics device. SPOTFIRE S+ therefore assumes there are no plots on the device. In this case, a call to a high-level plotting function does not erase the canvas before displaying a new plot. As soon as a high-level graphics function is called, `new` is set to `FALSE`. In this case, high-level graphics functions such as `plot` move to the next figure, or erase the current figure if there is only one, to avoid overwriting a plot.

You can take advantage of the new `graphics` parameter to call two high-level plotting functions in succession without erasing the first plot. The code below illustrates how to use the new parameter to overlay two plots that have the same x axis but different y axes. We first set `mar` so there is room for a labeled axis on both the left and right sides of the figure, then produce the first plot and the legend.

```
> par(mar = c(5,4,4,5) + 0.1)
> plot(hstart, ylab = "Housing Starts", type = "l")
> legend(1966.3, 220,
+       c("Housing Starts", "Manufacturing Shipments"),
+       lty = 1:2)
```

Now we set `new` to `TRUE` so that the first plot is not erased by the second. We also specify direct axes for the x axis in the second plot:

```
> par(new = T, xaxs = "d")
> plot(ship, axes = F, lty = 2, type = "l", ylab = "")
> axis(side = 4)
> mtext(side = 4, line = 2.75,
+       "Manufacturing (millions of dollars)")

# Release the direct axis.
> par(xaxs="r")
```

The subplot Function

The `subplot` function is another way to overlay plots with different scales. The `subplot` function allows you to put any Spotfire S+ graphic (except those created by `brush` and `spin`) into another graphic. To use `subplot`, specify the graphics function and the coordinates of the subplot on the current device. As an example, the code below produces a plot showing selected cities in New England, as well as New England's position relative to the rest of the United States. To achieve this figure, `subplot` is called several times.

To create the main plot, use the `usa` function with the arguments `xlim` and `ylim` to restrict attention to New England:

```
> usa(xlim = c(-72.5, -65), ylim = c(40.4, 47.6))
```

The coordinates in this command were obtained by trial-and-error, using the coordinates of New York as a starting point. The coordinates of New York were obtained from the three built-in data sets `city.x`, `city.y`, and `city.name`.

Before `city.x` or `city.y` can be used as an argument to a replacement function, it must first be assigned locally:

```
> city.x <- city.x
> city.y <- city.y
> names(city.x) <- city.name
> names(city.y) <- city.name
> nyc.coord <- c(city.x["New York"], city.y["New York"])
> nyc.coord

New York New York
-73.9667  40.7833
```

To plot the city names, we first use `city.x` and `city.y` to determine which cities are contained in the plotted area:

```
> ne.cities <- city.x > -72.5 & city.y > 40.4
```

We then use this criterion to select cities to label:

```
> text(city.x[ne.cities], city.y[ne.cities],
+      city.name[ne.cities])
```

For convenience in placing the subplot, retrieve the `usr` coordinates:

```
> usr <- par("usr")
```

Now we create a subplot of the United States and save the value of this call so that information can be added to it:

```
> subpars <- subplot(x = c(-69,usr[2]), y = c(usr[3],43),
+                   usa(xlim = c(-130,-50)))
```

The rest of the commands add to the small map of the entire United States. First, draw a box around the small US map:

```
> subplot(box(), pars = subpars)
```

Next, draw a box around New England:

```
> subplot(polygon(c(usr[1], -65, -65, usr[1]),
+                 c(usr[3], usr[3], usr[4], usr[4]), density = 0),
+         pars = subpars)
```

Finally, add text to indicate that the boxed region just created corresponds to the enlarged region:

```
> subplot(text((usr[1] + usr[2])/2, usr[4] + 4,
+             "Enlarged Region"), pars = subpars)
```

The `subplot` function can also be used to create *composite* figures. For example, the code below plots density estimates of the marginal distributions of mileage and price in the margins of a scatter plot of the two variables.

First, we set up the coordinate system with `par` and `usr`, and create and store the main plot with `subplot`:

```
> frame()
> par(usr = c(0,1,0,1))
> o.par <- subplot(x = c(0, 0.85), y = c(0, 0.85),
+   fun = plot(price, mileage, log = "x"))
```

Next, we find the `usr` coordinates from the main plot and calculate the density estimate for both variables:

```
> o.usr <- o.par$usr
> den.p <- density(price, width = 3000)
> den.m <- density(mileage, width = 10)
```

Finally, we plot the two marginal densities with successive calls to `subplot`. The first call plots the density estimate for price along the top of the main plot:

```
> subplot(x = c(0, 0.85), y = c(0.85, 1),
+ fun = {
+   par(usr = c(o.usr[1:2], 0, 1.04*max(den.p$y)),
+     xaxt = "l")
+   lines(den.p)
+   box()
+   }
+ )
```

The `xaxt="l"` graphics parameter (or, for R compatibility, `xlog=T`) is necessary in the first marginal density plot, since price is plotted with a logarithmic axis. To plot the density estimate for mileage along the right side of the main plot, use `subplot` as follows:

```
> subplot(x = c(0.85, 1), y = c(0, 0.85),
+ fun = {
+   par(usr = c(0, 1.04*max(den.m$y), o.usr[3:4]))
+   lines(den.m$y,den.m$x)
+   box()
+   }
+ )
```

ADDING SPECIAL SYMBOLS TO PLOTS

In the section Interactively Adding Information to Your Plot (page 66), we saw how to add lines and new data to existing plots. In this section, we describe how to add arrows, stars, and other special symbols to plots.

Arrows and Line Segments

To add one or more arrows to an existing plot, use the `arrows` function. To add a line segment, use the `segments` function. Both `segments` and `arrows` take beginning and ending coordinates, so that one or more line segments are drawn on the plot. For example, the following commands plot the `corn.rain` data and draw arrows from observation i to observation $i+1$:

```
> plot(corn.rain)
> for (i in seq(along = corn.rain))
+   arrows(1889+i, corn.rain[i], 1890+i, corn.rain[i+1],
+   size=0.2)
```

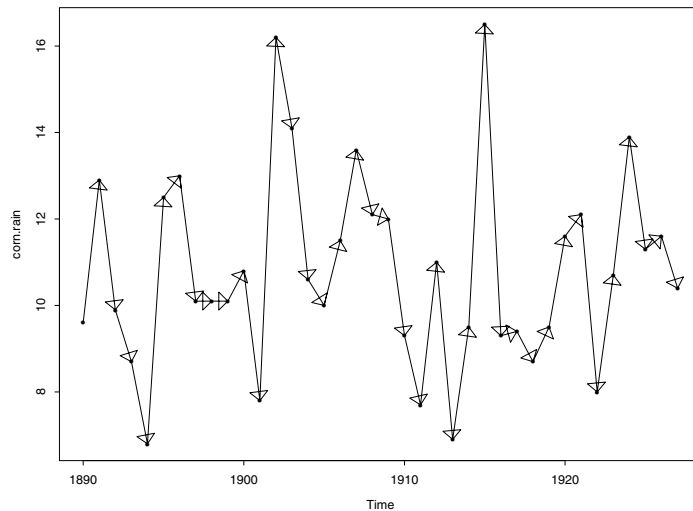


Figure 2.32: Adding arrows to plots using the `arrows` function.

Note

The `plot` function, and functions that add arrows and line segments to a plot, accept a vector of values for the `col`, `lty`, and `lwd` parameters. For example:

```
plot(exch.rate, lwd = 1:4)
```

For more information, see the section *Vectorized Graphics Parameters* (page 21) and the `par` help file.

Use the `segments` function similarly:

```
# Redefine x and y vectors introduced
# earlier in this chapter.
> set.seed(12)
> x <- runif(12)
> y <- rnorm(12)

# Plot the data with line segments.
> plot(x,y)
> for (i in seq(along = x))
+   segments(x[i], y[i], x[i+1], y[i+1])
```

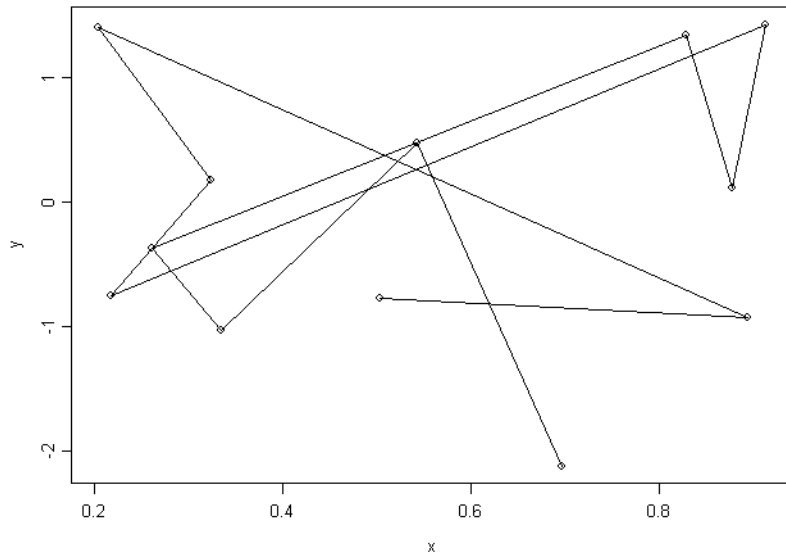


Figure 2.33: Adding line segments to plots using the `segments` function.

Stars and Other Symbols

You can display a third dimension of data in your plots by using the `symbols` function, which encodes data as stars, circles, or other special symbols. As an example, the steps below describe how to plot cities on a map of the United States with circles that have areas representing populations.

First, we create the data by selecting twelve cities from the built-in data set `city.name`:

```
> select <- c("Atlanta", "Atlantic City", "Bismarck",  
+ "Boise", "Dallas", "Denver", "Lincoln", "Los Angeles",  
+ "Miami", "Milwaukee", "New York", "Seattle")
```

As described in the section *Overlaying Figures* (page 97), we use the `names` function to assign the city names as vector names for the data sets `city.x`, `city.y`, and `city.name`. Before `city.x`, `city.y`, or `city.name` can be used as an argument to a replacement function, however, it must be assigned to your local working directory:

```
> city.x <- city.x  
> city.y <- city.y  
> city.name <- city.name  
> names(city.x) <- city.name  
> names(city.y) <- city.name  
> names(city.name) <- city.name
```

By assigning names in this way, we can access the information necessary to plot the cities without learning their vector indices. From an almanac or similar reference, we look up the populations of the selected cities and create a vector to hold the information:

```
> pop <- c(471, 40, 57, 193, 1214, 558, 239, 3845, 386, 579,  
+ 8143, 574)
```

The units of `pop` are in thousands. Use the `usa` function to plot a map of the United States:

```
> usa()
```

Next, add the circles representing the cities:

```
> symbols(city.x[select], city.y[select],  
+ circles = sqrt(pop), add = T)
```

The next two commands use the `ifelse` function to create a size vector for controlling the text size:

```
> size <- ifelse(pop>1000, 2, 1)
> size <- ifelse(pop<100, 0.5, size)
```

Taken together, these two lines specify a size of 2 for cities with populations greater than one million, a size of 1 for cities with populations between one hundred thousand and one million, and a size of 0.5 for cities with populations less than one hundred thousand. Finally, we add the text, using the `size` vector to specify the text size:

```
> text(city.x[select], city.y[select], city.name[select],
+      cex = size)
```

You can use any one of the shapes listed in Table 2.9 as an argument to `symbols`, with values as indicated.

Table 2.9: *Using shapes as an argument to the function `symbols`.*

Shape	Values
<code>circles</code>	Vector or matrix with one column containing the radii of the circles.
<code>squares</code>	Vector or matrix with one column containing the lengths of the sides of the squares.
<code>rectangles</code>	Matrix with two columns giving widths and heights of rectangles.
<code>stars</code>	Matrix with n columns, where n is the number of points in a star. The matrix must be scaled from 0 to 1.
<code>thermometers</code>	Matrix with 3 or 4 columns. The first two columns give the widths and heights of the rectangular thermometer symbols. If the matrix has 3 columns, the third column gives the fraction of the symbol that is filled (from the bottom up). If the matrix has 4 columns, the third and fourth columns give the fractions of the rectangle between which it is filled.
<code>boxplots</code>	Matrix with 5 columns of positive numbers, giving the width and height of the box, the amount to extend on the top and bottom, and the fraction of the box (from the bottom up) at which to draw the median line.

Missing values are allowed for all of these shapes; points containing missing values are not plotted. The one exception to this is `stars`, where missing values are treated as zeros.

Custom Symbols

The following two functions, `make.symbol` and `draw.symbol`, provide a simple way to add your own symbols to a plot.

The `make.symbol` function facilitates creating a new symbol:

```
> make.symbol <- function() {
+   on.exit(par(p))
+   p <- par(pty = "s")
+   plot(0, 0, type = "n", xlim = c(-0.5, 0.5),
+        ylim = c(-0.5, 0.5))
+   cat("Now draw your symbol using the mouse,
+       Continue string: clicking at corners\n ")
+   locator(type = "l")
+ }
```

SPOTFIRE S+ provides the `Continue` string: prompt because there is a new line in the middle of a character string; you do not need to type the prompt explicitly. The `make.symbol` function returns a list with components named `x` and `y`. The most important feature of this function is that it uses `pty="s"`, so the figure is drawn to the proper scale when used with `draw.symbol`. The `draw.symbol` function takes some locations to plot, and a symbol given in the form of a list with `x` and `y` components:

```
> draw.symbol <-
+ function(x, y, sym, size = 1, fill = F, ...) {
+   # inches per user unit
+   uin <- par()$uin
+   sym$x <- sym$x/uin[1]*size
+   sym$y <- sym$y/uin[2]*size
+   if (!fill)
+     for(i in 1:length(x))
+       lines(x[i]+sym$x, y[i]+sym$y, ...)
+   else
+     for(i in 1:length(x))
+       polygon(x[i]+sym$x, y[i]+sym$y, ...)
+ }
```

The `uin` graphics parameter is used to scale the symbol into user units. You can then plot your custom symbol with commands similar to the following:

```
> my.symbol <- make.symbol()  
Now draw your symbol using the mouse, clicking at corners  
  
> draw.symbol(0, 0, my.symbol)
```

The `make.symbol` and `draw.symbol` functions are examples of how you can create your own graphics functions using the built-in functions and parameters.

TRADITIONAL GRAPHICS SUMMARY

Table 2.10: Summary of the most useful graphics parameters.

Name	Type	Mode	Description	Example
MULTIPLE FIGURES				
<code>fig</code>	layout	numeric	figure location	<code>c(0, 0.5, 0.3, 1)</code>
<code>fin</code>	layout	numeric	figure size	<code>c(3.5,4)</code>
<code>fty</code>	layout	character	figure type	<code>"r"</code>
<code>mfg</code>	layout	integer	location in figure array	<code>c(1,1,2,3)</code>
<code>mfc0l</code>	layout	integer	figure array size	<code>c(2,3)</code>
<code>mfm0w</code>	layout	integer	figure array size	<code>c(2,3)</code>
TEXT				
<code>adj</code>	general	numeric	text justification	<code>0.5</code>
<code>cex</code>	general	numeric, vector	font height	<code>1.5</code> <code>rep(1:3, each=20)</code>
<code>cex.axis</code>	general	numeric	font height of axis annotation	<code>1.5</code>
<code>cex.lab</code>	general	numeric	font height of x and y labels	<code>1.5</code>
<code>cex.main</code>	general	numeric	font height of main title	<code>1.5</code>
<code>cex.sub</code>	general	numeric	font height of subtitle	<code>1.5</code>

Table 2.10: Summary of the most useful graphics parameters. (Continued)

Name	Type	Mode	Description	Example
<code>crt</code>	general	numeric	character rotation	90
<code>csi</code>	general	numeric	height of font	0.11
<code>font</code>	general	integer, vector	typeface	2 1:3
<code>font.axis</code>	general	integer	typeface of axis annotation	2
<code>font.lab</code>	general	integer	typeface of x and y labels	2
<code>font.main</code>	general	integer	typeface of main title	2
<code>font.sub</code>	general	integer	typeface of subtitle	2
<code>main</code>	title	character	main title	"Y versus X"
<code>srt</code>	general	numeric	string rotation	90
<code>sub</code>	title	character	subtitle	"Y versus X"
<code>xlab</code>	title	character	axis titles	"X (in dollars)"
<code>ylab</code>	title	character	axis title	"Y (in size)"
SYMBOLS				
<code>lty</code>	general	integer, vector	line type	2 1:4
<code>lwd</code>	general	numeric, vector	line width	3 1:4

Table 2.10: Summary of the most useful graphics parameters. (Continued)

Name	Type	Mode	Description	Example
<code>pch</code>	general	character, integer, vector	plot symbol	<code>"*"</code> , 4, <code>rep(0:2,each=20)</code>
<code>smo</code>	general	integer	curve smoothness	1
<code>type</code>	general	character	plot type	<code>"h"</code>
<code>xpd</code>	general	logical	symbols in margins	TRUE
AXES				
<code>axes</code>	high-level	logical	plot axes	FALSE
<code>bty</code>	general	character	box type	<code>"7"</code> , <code>"c"</code> , <code>"o"</code> (default)
<code>exp</code>	general	numeric	format for exponential numbers	1
<code>lab</code>	general	integer	tick marks and labels	<code>c(3,7,4)</code>
<code>las</code>	general	integer	label orientation	1
<code>log</code>	high-level	character	logarithmic axes	<code>"xy"</code>
<code>mgp</code>	general	numeric	axis locations	<code>c(3,1,0)</code>
<code>tck</code>	general	numeric	tick mark length	0.1
<code>xaxs</code>	general	character	style of limits	<code>"i"</code>
<code>yaxs</code>	general	character	style of limits	<code>"i"</code>
<code>xaxt</code>	general	character	axis type	<code>"n"</code>

Table 2.10: Summary of the most useful graphics parameters. (Continued)

Name	Type	Mode	Description	Example
<code>yaxt</code>	general	character	axis type	"n"
MARGINS				
<code>mai</code>	layout	numeric	margin size	<code>c(0.4, 0.5, 0.6, 0.2)</code>
<code>mar</code>	layout	numeric	margin size	<code>c(3,4,5,1)</code>
<code>mex</code>	layout	numeric	margin units	0.5
<code>oma</code>	layout	numeric	outer margin size	<code>c(0,0,5,0)</code>
<code>omd</code>	layout	numeric	outer margin size	<code>c(0,.95,0,1)</code>
<code>omi</code>	layout	numeric	outer margin size	<code>c(0,0,.5,0)</code>
PLOT REGION				
<code>pin</code>	layout	numeric	plot region	<code>c(3.5,4)</code>
<code>plt</code>	layout	numeric	plot region	<code>c(0.05,0.95,0.1,0.9)</code>
<code>pty</code>	layout	character	plot type	"s"
<code>uin</code>	information	numeric	inches per user unit	<code>c(0.73, 0.05)</code>
<code>usr</code>	layout	numeric	limits in plot region	<code>c(76,87,3,8)</code>
<code>xlim</code>	high-level	numeric	limits in plot region	<code>c(3,8)</code>
<code>ylim</code>	high-level	numeric	limits in plot region	<code>c(3,8)</code>

Table 2.10: Summary of the most useful graphics parameters. (Continued)

Name	Type	Mode	Description	Example
MISCELLANEOUS				
<code>col</code>	general	character, integer, vector	color of plot content	<code>"red", "blue"</code> <code>2,</code> <code>rep(c("red","green",</code> <code>"blue"),each=20)</code>
<code>col.axis</code>	general	character, integer	color of axis labels	<code>"red", "blue"</code> <code>2</code>
<code>col.lab</code>	general	character, integer	color of x and y labels	<code>"red", "blue"</code> <code>2</code>
<code>col.main</code>	general	character, integer	color of main title	<code>"red", "blue"</code> <code>2</code>
<code>col.sub</code>	general	character, integer	color of subtitle	<code>"red", "blue"</code> <code>2</code>
<code>fg</code>	general	character, integer	foreground color	<code>"red", "blue"</code> <code>2</code>
<code>bg</code>	general	character, integer	background color	<code>"red", "blue"</code> <code>2</code>
<code>err</code>	general	integer	print warnings?	<code>-1</code>
<code>new</code>	layout	logical	is figure blank?	<code>TRUE</code>

REFERENCES

- Chernoff, H. (1973). The Use of Faces to Represent Points in k-Dimensional Space Graphically. *Journal of American Statistical Association*, **68**: 361-368.
- Cleveland, W.S. (1985). *The Elements of Graphing Data*. Wadsworth: Monterrey, California.
- Martin, R.D., Yohai, V.J., and Zamar, R.H. (1989). Min-max bias robust regression. *Annals of Statistics*, **17**: 1608-30.
- Silverman, B.W. (1986). *Density Estimation for Statistics and Data Analysis*. London: Chapman and Hall.
- Venables, W.N. and Ripley, B.D. (1999). *Modern Applied Statistics with S-PLUS* (3rd edition). New York: Springer-Verlag.

TRADITIONAL TRELLIS GRAPHICS

3

A Roadmap of Trellis Graphics	117
Getting Started	117
General Display Functions	117
Common Arguments	117
Panel Functions	117
Core Spotfire S+ Graphics	118
Printing, Devices, and Settings	118
Data Structures	119
Giving Data to Trellis Functions	120
The formula and data Arguments	120
The subset Argument	122
General Display Functions	124
Scatter Plots: the xyplot Function	124
Visualizing One-Dimensional Data	125
Visualizing Two-Dimensional Data	133
Visualizing Three-Dimensional Data	136
Visualizing Multi-Dimensional Data	141
Summary: The Display Functions and Their Formulas	144
Arranging Several Graphs on One Page	146
Multipanel Conditioning	148
About Multipanel Display	148
Columns, Rows, and Pages	149
Packet Order and Panel Order	151
Main-Effects Ordering	153
Conditioning on the Values of a Numeric Variable	155
Summary: The Layout of a Multipanel Display	161
General Options for Multipanel Displays	162
Spacing Between Rows and Columns	162
Skipping Panels	164
Multipage Displays	164

Scales and Labels	166
Axis Labels and Titles	166
Axis Limits	167
Tick Marks and Labels	167
Changing the Text in Strip Labels	168
Panel Functions	171
Passing Arguments to a Default Panel Function	171
Writing a Custom Panel Function	172
Special Panel Functions	173
Summary: Common Options in Panel Functions	174
Panel Functions and the Trellis Settings	175
The <code>trellis.par.get</code> Function	175
The <code>show.settings</code> Function	177
The <code>trellis.par.set</code> Function	177
Superposing Multiple Value Groups on a Panel	179
Superposing Points	179
Superposing Curves	181
Superposing Other Plots	182
The <code>key</code> Argument	183
Aspect Ratio	189
2D Displays	189
3D Displays	190
Prepanel Functions	191
Data Structures	194
Vectors	194
Arrays	195
Time Series	196
Summary of Trellis Functions and Arguments	198

A ROADMAP OF TRELIS GRAPHICS

Trellis Graphics provide a comprehensive set of display functions that are a popular alternative to the functions described in Chapter 2, Traditional Graphics. The Trellis functions are particularly geared towards multipanel and multipage plots. This chapter describes the Trellis system based on traditional Spotfire S+ graphics.

Getting Started

You can open a Trellis Graphics device with the command `trellis.device`:

```
> trellis.device()
```

If no device is open, Trellis functions open one by default; however, explicitly calling `trellis.device` ensures that the open graphics device is compatible with Trellis Graphics.

General Display Functions

The Trellis library has a collection of *general display functions* that draw different types of graphs. For example, `xyplot` displays *xy* plots, `dotplot` displays dot plots, and `wireframe` displays three-dimensional wireframe plots. The functions are *general* because they have the full capability of Trellis Graphics, including multipanel conditioning. These functions are introduced in the section General Display Functions (page 124).

Common Arguments

All general display functions share a common set of arguments. The usage of these arguments varies from function to function, but each has a common purpose. Many of the general display functions also have arguments that are specific to the types of graphs that they draw.

The common arguments, which are listed in the section Summary of Trellis Functions and Arguments (page 198), are discussed in many of the sections throughout this chapter.

Panel Functions

Panel functions are a critical aspect of Trellis Graphics. They make it easy to tailor displays to your data, even when the displays are quite complicated and have many panels.

The *data region* of a panel in a Trellis graph is a rectangle that just encloses the data. Panel functions have sole responsibility for drawing in data regions; they are specified by a `panel` argument to the general display functions. Panel functions manage the symbols, lines, and so forth that encode the data in the data regions. The other arguments to the general display functions manage the superstructure of the graph, such as scales, labels, boxes around the data region, and keys.

Panel functions are discussed in the section Panel Functions (page 171).

Core Spotfire S+ Graphics

Trellis Graphics is implemented using the core Spotfire S+ graphics discussed in Chapter 2, Traditional Graphics. In addition, you use functions and graphics parameters from the traditional graphics system when you write custom panel functions. Some of these graphics features are discussed in the section Summary: Common Options in Panel Functions (page 174).

Printing, Devices, and Settings

To print a Trellis graph, first open a hardcopy device using the `trellis.device` function. For example, the following command opens a `pdf.graph` device using the default Trellis formats:

```
> trellis.device(pdf.graph, file = "mygraph.pdf")
```

To send the graphics to the printer, enter the command `dev.off()`; this writes the graphics to the file **mygraph.pdf**, which you can then view and print with Adobe Acrobat Reader. Alternatively, you can choose **File ► Print** while the **Graph** window is active in those versions of Spotfire S+ that include a graphical user interface.

By default, graphics are printed in black and white. For color printing, set the flag `color=TRUE` when you open the graphics device. For example, the following command opens a color postscript device:

```
> trellis.device(postscript, file="mycolor.ps", color=TRUE)
```

The Trellis library has many settings for graph rendering details, including plotting symbols, colors, and line types. These settings are automatically chosen depending on the device you select. The section Panel Functions and the Trellis Settings (page 175) discusses the Trellis settings in more detail.

**Data
Structures**

The general display functions accept data just like many of the Spotfire S+ modeling functions (`lm`, `aov`, `glm`, and `loess`, for example). This means that there is a heavy reliance on data frames. You can keep variables as vectors and draw Trellis displays without using data frames, but data frames are nevertheless convenient. The Trellis library contains several functions that change data structures of certain types to data frames. These functions are discussed in the section Data Structures (page 194).

GIVING DATA TO TRELLIS FUNCTIONS

This section describes the arguments to the Trellis Graphics functions that allow you to specify the data sets and variables to be drawn. In the examples of this section, we use the built-in data set `gas`, which contains two variables from an industrial experiment with twenty-two runs. In the experiment, the concentrations of oxides of nitrogen (`NOx`) in the exhaust of an engine were measured for different settings of equivalence ratio (`E`).

```
> names(gas)
[1] "NOx" "E"

> dim(gas)
[1] 22 2
```

The formula and data Arguments

The function `xypLOT` draws an *xy* plot, which is a graph of two numerical variables; the resulting plot might be scattered points, curves, or both. A full discussion of `xypLOT` is in the section General Display Functions (page 124), but for now we use it to illustrate how to specify data.

The plot in Figure 3.1, generated by the following command, is a scatter plot of `gas$NOx` against `gas$E`:

```
> xypLOT(formula = gas$NOx ~ gas$E)
```

The argument `formula` specifies the variables to be graphed. For `xypLOT`, the variable to the left of the tilde (`~`) is plotted on the vertical axis and the variable to the right is plotted on the horizontal axis. You can read the formula `gas$NOx~gas$E` as “`gas$NOx` is graphed against `gas$E`”.

The use of `formula` in Trellis display functions is the same as the use in statistical modeling functions such as `lm` and `aov`. To the left or right of the tilde, you can use any Spotfire S+ expression. For example, if you want to graph the base 2 logarithm of `gas$NOx`, you can use the formula in the following command:

```
> xypLOT(formula = logb(gas$NOx, base=2) ~ gas$E)
```

The formula argument is a special one in Trellis Graphics. It is always the first argument of a general display function such as `xyplot`. We can therefore omit typing `formula` explicitly when we call a general display function, provided the formula is the first argument. Thus the expression `xyplot(gas$NOx ~ gas$E)` also produces Figure 3.1.

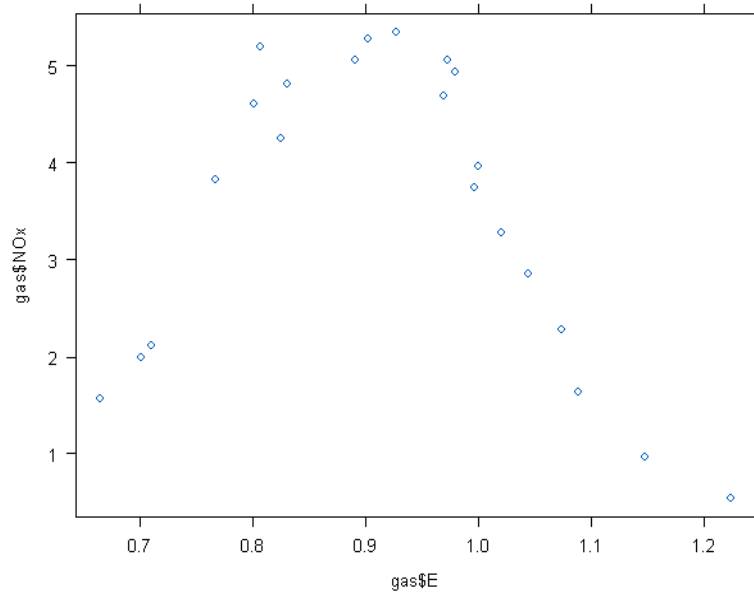


Figure 3.1: Scatter plot of `gas$NOx` against `gas$E`.

Certain operators that perform functions in Spotfire S+ have special meanings in the formula language (for example, `+`, `*`, `/`, `|`, and `:`). Trellis formulas, as we see throughout in this chapter, use only the `*` and `|` operators. If you want to use a special operators for its conventional meaning in a formula, wrap the expression in a call to the identity function `I`. For example, the following command uses `*` as multiplication:

```
> xyplot(logb(2*gas$NOx, base=2) ~ I(2*gas$E))
```

We use `I` on the right side of the formula to protect against the `*` in `2*gas$E`, but we do not need `I` on the left because `2*gas$NOx` sits inside the function `logb`.

In the above commands, we continually refer to the data frame `gas` and subscript the columns explicitly. This is not necessary if we attach `gas` to the search list of databases. For example, we can draw Figure 3.1 with the following commands:

```
> attach(gas)
> xyplot(N0x~E)
> detach("gas")
```

Another possibility is to use the argument `data`:

```
> xyplot(N0x~E, data = gas)
```

In this case, the variables in `gas` are available for use in the `formula` argument during the execution of `xyplot`; the effect is the same as using `attach` and `detach`.

The use of the `data` argument has another benefit: in the call to `xyplot`, we see clearly that the data frame `gas` is being used. This can be helpful in understanding how the graph was produced at some future point in time.

The subset Argument

Suppose you want to redo Figure 3.1 and omit the observations for which `E` is 1.1 or greater. You could accomplish this with the following command:

```
> xyplot(N0x[E < 1.1] ~ E[E < 1.1], data = gas)
```

However, it is a nuisance to repeat the logical subsetting `E < 1.1`, and the nuisance is much greater when there are many variables in the formula instead of just two. It is typically easier to use the `subset` argument instead:

```
> xyplot(N0x~E, data = gas, subset = E < 1.1)
```

The result is shown in Figure 3.2. The subset argument can be a logical vector as in this example, or it can be numerical vector specifying the row numbers of data to plot.

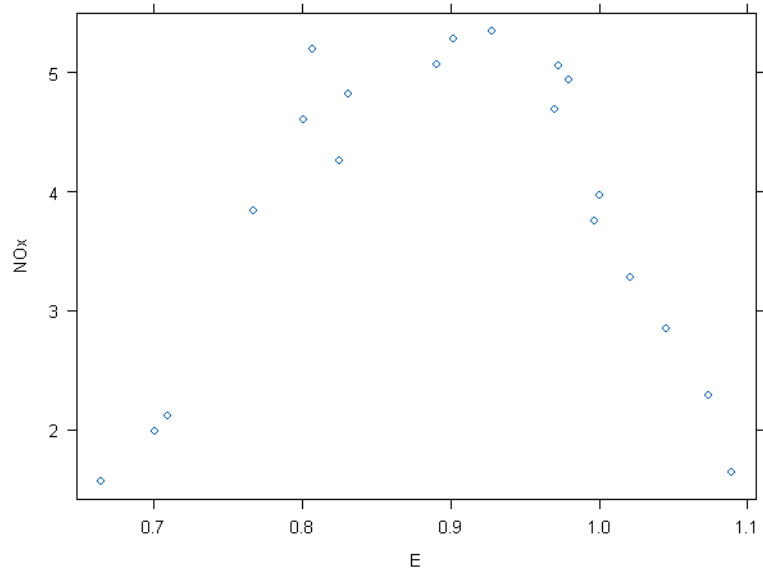


Figure 3.2: *Using the subset argument on the gas data.*

GENERAL DISPLAY FUNCTIONS

Each *general display function* in the Trellis library draws a particular type of graph. For example, `dotplot` creates dot plots, `wireframe` creates three-dimensional wireframe displays, `histogram` creates histograms, and `xyplot` creates *xy* plots. This section describes a collection of general display functions. Many of the examples in this section use the optional `aspect` argument to set the aspect ratio of the plots; for more details on the `aspect` argument, see the section *Aspect Ratio* (page 189).

In many of our examples, we use the built-in data set `fuel.frame`, which contains five variables that measure characteristics of 60 automobile models:

```
> names(fuel.frame)
[1] "Weight" "Disp." "Mileage" "Fuel" "Type"

> dim(fuel.frame)
[1] 60 5
```

The variables are weight, engine displacement, fuel consumption in miles per gallon, fuel consumption in gallons per mile, and a classification of the type of vehicle. The first four variables are numeric and the fifth is a factor:

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

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

Scatter Plots: the `xyplot` Function

We have already seen the `xyplot` function in the examples of the previous section. This function is a basic graphical method, displaying one set of numerical values on a vertical scale against another set on a horizontal scale. For example, Figure 3.3 is a scatter plot of mileage against weight using the `fuel.frame` data:

```
> xyplot(Mileage~Weight, data=fuel.frame, aspect=1)
```

The variable on the left of the `~` goes on the vertical axis and the variable on the right goes on the horizontal axis.

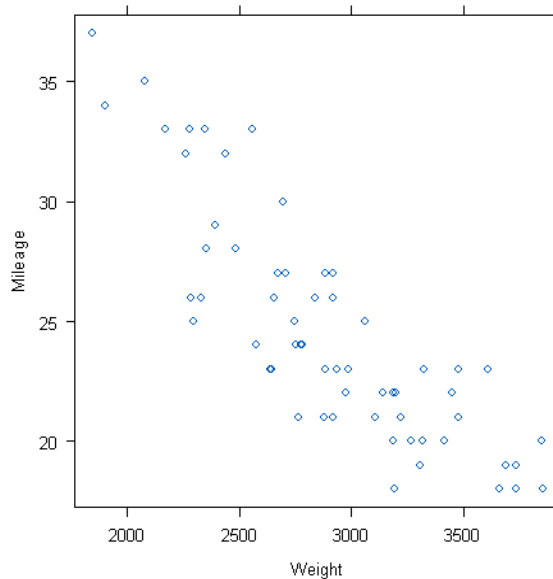


Figure 3.3: Scatter plot of the mileage and weight data in `fuel.frame` using `xypLOT`.

Visualizing One- Dimensional Data

A *one-dimensional data object* is sometimes referred to as a (single) *data sample*, a set of *univariate observations*, or simply a batch of data. In this section, we examine a number of basic plot types useful for exploring a one-dimensional data object.

- **Density Plot:** an estimate of the underlying probability density function for a data set.
- **Histogram:** a display of the number of data points that fall in each of a specified number of intervals. A histogram gives an indication of the relative density of the data points along the horizontal axis.
- **QQ Math Plot:** an extremely powerful tool for determining a good approximation to a data set's distribution. The most common is the *normal probability plot*, or *normal qqplot*, which is used to test whether the distribution of a data set is nearly Gaussian.

- **Bar Chart:** a display of the relative magnitudes of observations in a data set.
- **Dot Plot:** a tool that displays the same information as a bar chart or pie chart, but in a form that is often easier to grasp.
- **Pie Chart:** a graph that shows the share of individual values in a variable, relative to the sum total of all the values.

These visualization plots are simple but powerful exploratory data analysis tools that can help you quickly grasp the nature of your data. Such an understanding can help you avoid the misuse of statistical inference methods, such as using a method appropriate only for a normal (Gaussian) distribution when the distribution is strongly non-normal.

Density Plots

As a first step in analyzing one-dimensional data, it is often useful to study the shape of its distribution. A *density plot* displays an estimate of the underlying probability density function for a data set, and allows you to approximate the probability that your data fall in any interval. The Trellis function that displays densities is called `densityplot`.

In Spotfire S+, density plots are essentially *kernel smoothers*. In this type of algorithm, a smoothing window is centered on each x value, and the predicted y value in the density plot is calculated as a weighted average of the y values for nearby points. The size of the smoothing window is called the *bandwidth* of the smoother. Increasing the bandwidth results in a smoother curve but may miss rapidly changing features. Decreasing the bandwidth allows the smoother to track rapidly changing features more accurately, but results in a rougher curve fit. Use the `width` argument in `densityplot` to vary the bandwidth value in your displays.

The weight given to each point in a smoothing window decreases as the distance between its x value and the x value of interest increases. *Kernel functions* specify the way in which the weights decrease: kernel choices for `densityplot` include a cosine curve, a normal (Gaussian) kernel, a rectangle, and a triangle. The default kernel is Gaussian, where the weights decrease with a normal (Gaussian) distribution away from the point of interest. A rectangular kernel weighs each point within the smoothing window equally, and a triangular kernel has linearly decreasing weights. In a cosine kernel, weights decrease with a cosine curve away from the point of interest. Use the `window` argument in `densityplot` to vary the kernel function in your displays.

Figure 3.4 is a density plot of the mileage data in `fuel.frame`:

```
> densityplot(~Mileage, data=fuel.frame, aspect=1/2,
+   width=5)
```

The `width` argument controls the width of the smoothing window in the same units as the data; here, the units are in miles per gallon. The rug at the bottom of the density plot shows the unique x values in the data set.

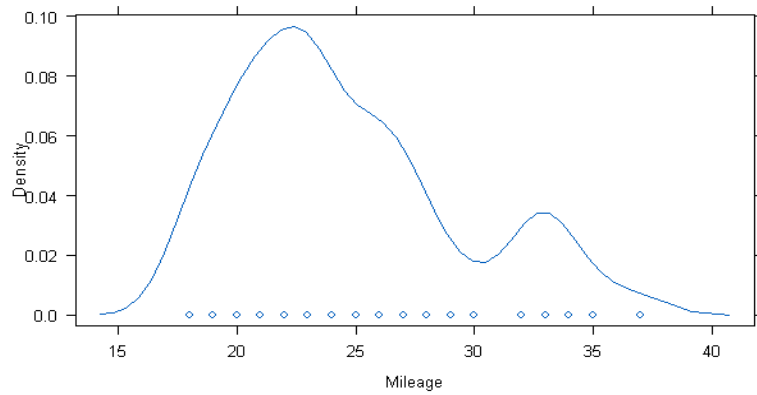


Figure 3.4: *Density plot of the mileage data in `fuel.frame` using the `densityplot` function.*

Histograms

Histograms display the number of data points that fall in each of a specified number of intervals. A histogram gives an indication of the relative density of the data points along the horizontal axis. For this reason, density plots are often superposed with (scaled) histograms. The Trellis function that displays histograms is called `histogram`.

Figure 3.5 is a histogram of the mileage data in `fuel.frame`:

```
> histogram(~Mileage, data=fuel.frame, aspect=1, nint=10)
```

The argument `nint` determines the number of intervals in which the data is binned. The `histogram` algorithm chooses the intervals to make the bar widths “nice” numbers, while trying to make the number of intervals as close to `nint` as possible.

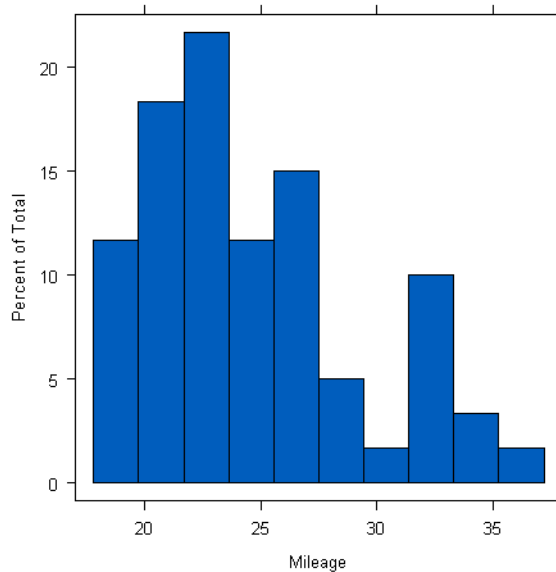


Figure 3.5: Histogram of the mileage data in `fuel.frame` using the `histogram` function.

QQ Math Plots

The quantile-quantile plot, or *qqplot*, is an extremely powerful tool for determining a good approximation to a data set's distribution. In a qqplot, the ordered data are graphed against quantiles of a known theoretical distribution. If the data points are drawn from the theoretical distribution, the resulting plot is close to the straight line $y=x$ in shape. The most common in this class of one-dimensional plots is the *normal probability plot*, or *normal qqplot*, which is used to test whether the distribution of a data set is nearly normal (Gaussian). One Trellis function that displays qqplots is called `qqmath`; see the section *Visualizing Two-Dimensional Data* (page 133) for a description of a second Trellis function.

Figure 3.6 is a normal probability plot of the mileage data for small cars:

```
> qqmath(~Mileage, data = fuel.frame,
+       subset = (Type=="Small"))
```

The `distribution` argument in `qqmath` governs the theoretical distribution used in the plot. It accepts Spotfire S+ functions that compute quantiles for theoretical distributions. By default, `distribution=qnorm` and normal probability plots are drawn. If instead we use the command

```
> qqmath(~Mileage, data = fuel.frame, aspect = 1
+ subset = (Type=="Small"), distribution = qexp)
```

the result is an exponential probability plot. Note that the name of the distribution function appears as the default label on the horizontal axis of the plot.

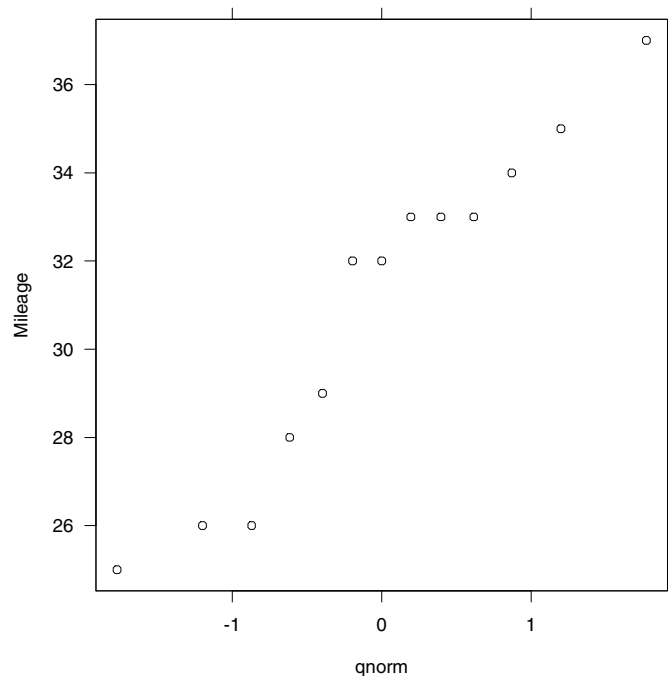


Figure 3.6: Normal probability plot of the mileage data for small cars using the `qqmath` function.

Bar Charts

A *bar chart* displays a bar for each point in a set of observations, where the height of a bar is determined by the value of the data point. The Trellis function that displays bar charts is called `barchart`.

As an example, we compute the mean mileage for each vehicle type in the `fuel.frame` data:

```
> mileage.means <- tapply(  
+   fuel.frame$Mileage, fuel.frame$Type, FUN=mean)
```

Figure 3.7 is a bar chart of the mileage means:

```
> barchart(names(mileage.means) ~ mileage.means, aspect=1)
```

Notice that the vehicle types in Figure 3.7 are ordered, from bottom to top, by the order of the elements of the vector `mileage.means`. This is determined by the order of the levels in the `Type` column:

```
> names(mileage.means)  
[1] "Compact" "Large" "Medium" "Small" "Sporty" "Van"  
  
> levels(fuel.frame$Type)  
[1] "Compact" "Large" "Medium" "Small" "Sporty" "Van"
```

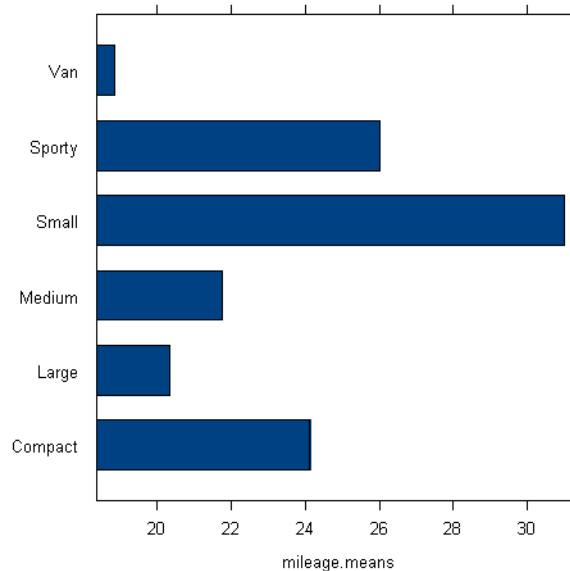


Figure 3.7: Bar chart of the `mileage.means` data using the `barchart` function.

Dot Plots

The dot plot was first described by Cleveland in 1985 as an alternative to bar charts and pie charts. The dot plot displays the same information as a bar chart or pie chart, but in a form that is often

easier to grasp. Instead of bars or pie wedges, dots and grid lines are used to mark the data values in dot plots. In particular, the dot plot reduces most data comparisons to straightforward length comparisons on a common scale. The Trellis function that displays dot plots is called `dotplot`.

Figure 3.8 is a dot plot of the base 2 logarithm of the `mileage.means` data created in the section Bar Charts:

```
> dotplot(names(mileage.means) ~
+         logb(mileage.means, base=2), aspect=1, cex=1.25)
```

Note that the vehicle categories appear on the vertical axis in the same order as they do in bar charts. The argument `cex` is passed to the panel function to change the size of the dot in the plot; for more information on panel functions, see the section Panel Functions (page 171).

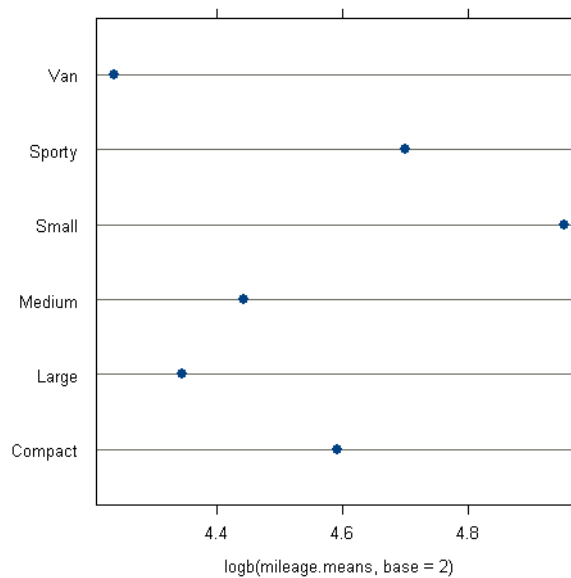


Figure 3.8: Dot plot of the `mileage.means` data using the `dotplot` function.

Pie Charts

A *pie chart* shows the share of individual values in a variable, relative to the sum total of all the values. Pie charts display the same information as bar charts and dot plots, but can be more difficult to interpret. This is because the size of a pie wedge is relative to a sum, and does not directly reflect the magnitude of the data value. Because

of this, pie charts are most useful when the emphasis is on an individual item's relation to the whole; in these cases, the sizes of the pie wedges are naturally interpreted as percentages. When such an emphasis is not the primary point of the graphic, a bar chart or a dot plot is preferred.

The Trellis function that displays pie charts is called `piechart`. For example, Figure 3.9 is a pie chart of the `mileage.means` data created in the section Bar Charts:

```
> piechart(names(mileage.means) ~ mileage.means)
```



Figure 3.9: Pie chart of the `mileage.means` data using the `piechart` function.

Because the average mileage of each type of car cannot be easily interpreted as a fraction of the total mileage, Figure 3.9 does not convey the information in `mileage.means` very well. We can see that small cars get slightly better mileage on average, since the corresponding pie wedge is the largest in the chart. Other than that, the size of the pie wedges simply imply that the mileage of the cars are relatively close in value when compared to the sum total. To refine these conclusions, we would need to view a bar chart or a dot plot of the data.

Visualizing Two- Dimensional Data

Two-dimensional data are often called *bivariate* data, and the individual, one-dimensional components of the data are referred to as *variables*. Two-dimensional plots help you quickly grasp the nature of the relationship between the two variables that constitute bivariate data. For example, you might want to know whether the relationship is linear or nonlinear, if the variables are highly correlated, if there are any outliers or distinct clusters, etc. In this section, we examine a number of basic plot types that are useful for exploring a two-dimensional data object.

- **Box Plot:** a graphical representation showing the center and spread of a distribution, as well as any outlying data points.
- **Strip Plot:** a one-dimensional scatter plot.
- **QQ Plot:** a powerful tool for comparing the distributions of two sets of data.

When you couple two-dimensional plots of bivariate data with one-dimensional visualizations of each variable's distribution, you gain a thorough understanding of your data.

Box Plots

A *box plot*, or box and whisker plot, is a clever graphical representation showing the center and spread of a distribution. A box is drawn that represents the bulk of the data, and a line or a symbol is placed in the box at the median value. The width of the box is equal to the *interquartile range*, or IQR, which is the difference between the third and first quartiles of the data. The IQR indicates the spread of the distribution for the data. Whiskers extend from the edges of the box to either the extreme values of the data, or to a distance of $1.5 \times \text{IQR}$ from the median, whichever is less. Data points that fall outside of the whiskers may be outliers, and are therefore indicated by additional lines or symbols.

The Trellis function that displays box plots is called `bwplot`. For example, Figure 3.10 is a box plot of mileage classified by vehicle type in the `fuel.frame` data:

```
> bwplot(Type~Mileage, data=fuel.frame, aspect=1)
```

Notice that the vehicle types in Figure 3.10 are ordered, from bottom to top, by the order of the levels in the `Type` column.

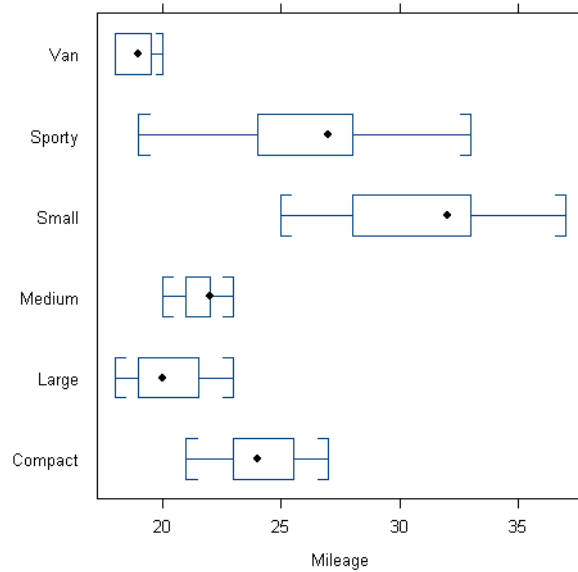


Figure 3.10: Box plot of the mileage data in `fuel.frame` using the `bwplot` function.

Strip Plots

A *strip plot* can be thought of as a one-dimensional scatter plot. Strip plots are similar to box plots in overall layout, but they display all of the individual data points instead of the box plot summary. The Trellis function that displays strip plots is called `stripplot`.

Figure 3.11 is a strip plot of the mileage data in `fuel.frame`:

```
> stripplot(Type~Mileage, data=fuel.frame, jitter=TRUE,  
+          aspect=1)
```

Setting the option `jitter=TRUE` causes some random noise to be added vertically to the points; this alleviates the overlap of the plotting symbols. By default, `jitter=FALSE` and the points for each level lie on a horizontal line.

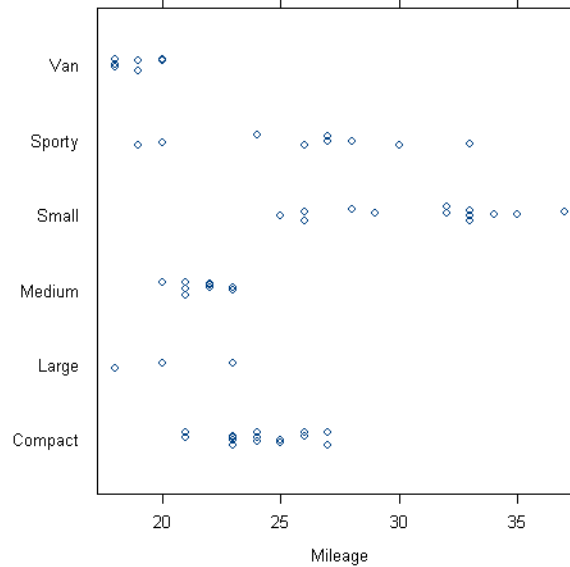


Figure 3.11: Strip plot of the mileage data in `fuel.frame` using the `stripplot` function.

QQ Plots

In the section *Visualizing One-Dimensional Data*, we introduced the quantile-quantile plot, or *qqplot*, as an extremely powerful tool for determining a good approximation to a data set's distribution. In a one-dimensional qqplot, the ordered data are graphed against quantiles of a known theoretical distribution. If the data points are drawn from the theoretical distribution, the resulting plot is close to the straight line $y=x$ in shape. We can also use qqplots with two-dimensional data to compare the distributions of the variables. In this case, the ordered values of the variables are plotted against each other. If the variables have the same distribution shape, the points in the qqplot cluster along the line $y=x$.

The Trellis function that displays two-dimensional qqplots is called `qq`. The `qq` function creates a qqplot for the two groups in a binary variable. It expects a numeric variable and a factor variable with exactly two levels; the values of the numeric variable corresponding to each level are then plotted against each other. For example, Figure 3.12 is a qqplot comparing the quantiles of mileage for compact cars with the corresponding quantiles for small cars:

```
> qq(Type~Mileage, data=fuel.frame, aspect=1,  
+     subset = (Type=="Compact") | (Type=="Small"))
```

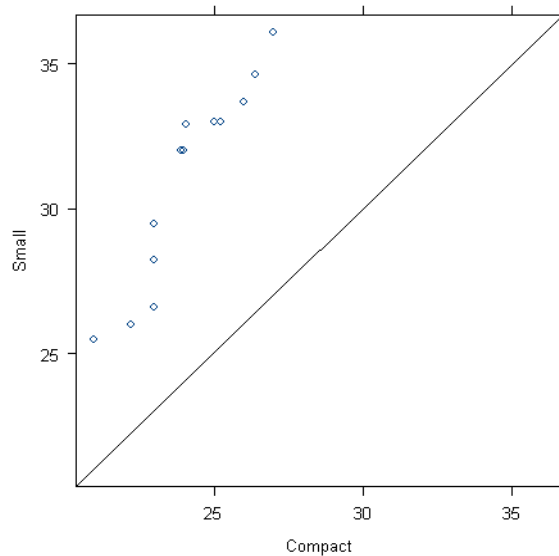


Figure 3.12: *QQplot of the mileage data for small and compact cars using the qq function.*

Visualizing Three- Dimensional Data

Three-dimensional data have three columns, or *variables*, of univariate data, and the relationships between variables form a surface in 3D space. Because the depth cues in three-dimensional plots are sometimes insufficient to convey all of the information, special considerations must be made when visualizing three-dimensional data. Instead of viewing the surface alone, we can analyze projections, slices, or rotations of the surface. In this section, we examine a number of basic plot types useful for exploring a three-dimensional data object.

- **Contour Plot:** uses contour lines to represent heights of three-dimensional data in a flat, two-dimensional plane.
- **Level Plot:** uses colors to represent heights of three-dimensional data in a flat, two-dimensional plane. Level plots and contour plots are essentially identical, but they have defaults that allow you to view a particular surface differently.

- **Surface Plot:** approximates the shape of a data set in three dimensions.
- **Cloud Plot:** displays a three-dimensional scatter plot of points.

In many of our examples in this section, we use the gauss data set, which consists of a function of two variables over a grid:

```
> datax <- rep(seq(from=-1.5, to=1.5, length=50), times=50)
> datay <- rep(seq(from=-1.5, to=1.5, length=50),
+   times = rep(50,times=50))
> dataz <- exp(-(datax^2 + datay^2 + datax*datay))
> gauss <- data.frame(datax, datay, dataz)
```

Thus, dataz is the exponential of a quadratic function defined over a 50 x 50 grid; in other words, the surface is proportional to a bivariate normal density.

Contour Plots

A *contour plot* is a representation of three-dimensional data in a flat, two-dimensional plane. Each contour line represents a height in the z direction from the corresponding three-dimensional surface. Contour plots are often used to display data collected on a regularly-spaced grid; if gridded data is not available, interpolation is used to fit and plot contours. The Trellis function that displays contour plots is called `contourplot`.

Figure 3.13 is a contour plot of the gauss data set:

```
> contourplot(dataz ~ datax*datay, data=gauss, aspect=1,
+   at = seq(from=0.1, to=0.9, by=0.2))
```

The argument `at` specifies the values at which the contours are computed and drawn. If no argument is specified, default values are chosen.

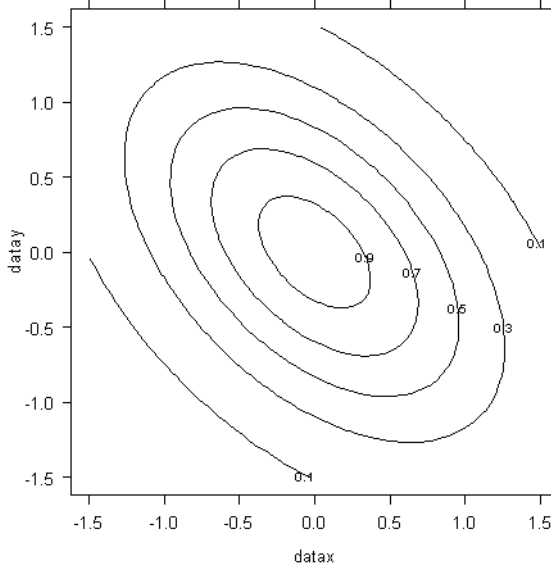


Figure 3.13: Contour plot of the gauss surface using the `contourplot` function.

Contour plots are helpful for displaying a function $f(x, y)$ when there is no need to study the conditional dependence of f on x given y , or of f on y given x . Conditional dependence is revealed far better by multipanel conditioning; for more details, see the section Multipanel Conditioning (page 148).

Level Plots

A *level plot* is essentially identical to a contour plot, but it has default options that allow you to view a particular surface differently. Like contour plots, level plots are representations of three-dimensional data in flat, two-dimensional planes. Instead of using contour lines to indicate heights in the z direction, however, level plots use colors. In general, level plots are no better than contour plots when the surface is simple, but they are often better when there is a lot of fine detail.

The Trellis function that displays level plots is called `levelplot`. For example, Figure 3.14 is a level plot of the gauss surface:

```
> levelplot(dataz ~ datax*datay, data=gauss, aspect=1,
+           cuts=6)
```

The values of the surface are encoded by color or gray scale. For devices with full color, the scale goes from pure magenta to white and then to pure cyan. If the device does not have full color, a gray scale is used.

For a level plot, the range of the function values is divided into intervals and each interval is assigned a color. A rectangle centered on each grid point is given the color of the interval containing the value of the function at the grid point. In Figure 3.14, there are six intervals. The argument `cuts` specifies the number of breakpoints between intervals.

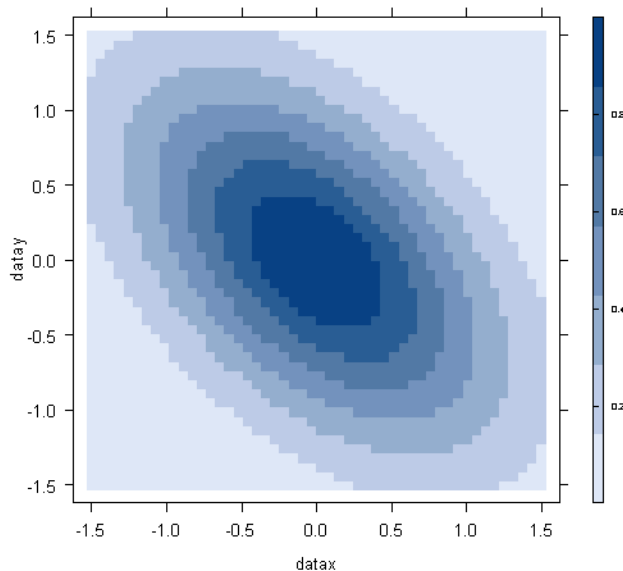


Figure 3.14: *Level plot of the gauss surface using the `levelplot` function.*

Surface Plots

A *surface plot* is an approximation to the shape of a three-dimensional data set. Surface plots are used to display data collected on a regularly-spaced grid; if gridded data is not available, interpolation is used to fit and plot the surface. The Trellis function that displays surface plots is called `wireframe`.

Figure 3.15 is a wireframe plot of the gauss surface:

```
> wireframe(dataz ~ datax*datay, data=gauss,
+   screen = list(z=45, x=-60, y=0))
```

The screen argument is a list with components x , y , and z that refer to screen axes. The surface is rotated about the axes in the order given in the list. Here is how the perspective in Figure 3.15 was created: the surface began with `datax` as the horizontal screen axis, `datay` as the vertical, and `dataz` as the perpendicular axis. The origin was at the lower left in the rear of the display. First, the surface was rotated 45° about the perpendicular screen axis, where a positive rotation is counterclockwise. Then the surface was rotated -60° about the horizontal screen axis, where a negative rotation pushes the top of the picture away from the viewer and pulls the bottom of the picture toward the viewer. Finally, there was no rotation about the vertical screen axis. However, if there had been a positive vertical rotation, the left side of the picture would have moved toward the viewer and the right side of the picture would have moved away.

If the argument `drape=TRUE`, color is added to the surface using the same encoding method of the level plot. By default, `drape=FALSE`.

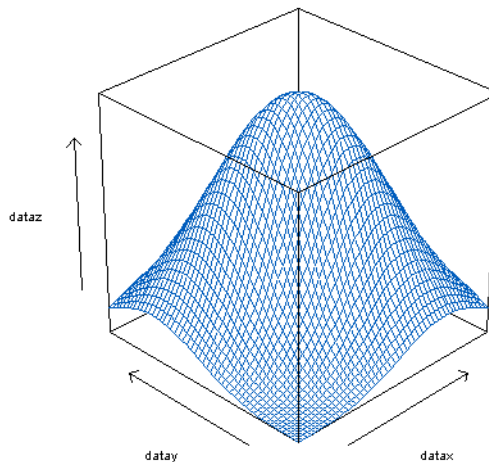


Figure 3.15: *Surface plot of the gauss data using the wireframe function.*

Cloud Plots

A *cloud plot* is a three-dimensional scatter plot of points. Typically, a static 3D scatter plot is not effective because the depth cues of single points are insufficient to give a strong 3D effect. On some occasions, however, cloud plots can be useful for discovering simple characteristics about the three variables. The Trellis function that displays cloud plots is called `cloud`.

Figure 3.16 is a 3D scatter plot of the first three variables in the data set `fuel.frame`:

```
> cloud(Mileage ~ Weight*Disp., data=fuel.frame,
+       screen = list(z=-30, x=-60, y=0),
+       xlab = "W", ylab = "D", zlab = "M")
```

The behavior of the `screen` argument is the same as that for `wireframe`. We have used the arguments `xlab`, `ylab`, and `zlab` to specify scale labels; such labeling is discussed in more detail in the section `Scales and Labels` (page 166).

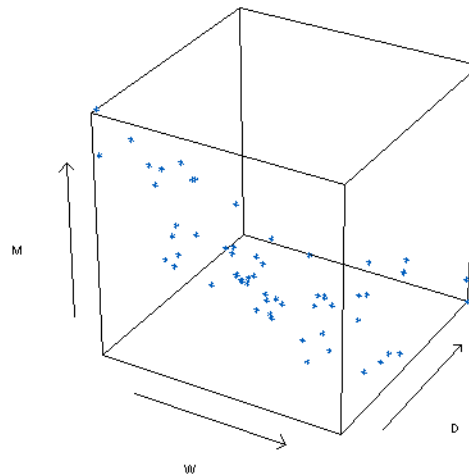


Figure 3.16: 3D scatter plot of the `fuel.frame` data using the `cloud` function.

Visualizing Multi- Dimensional Data

In the previous sections, we discussed visual tools for simple one-, two-, and three-dimensional data sets. With lower-dimensional data, all of the basic information in the data may be easily viewed in a single set of plots. Different plots provide different types of information, but deciding which plots to use is fairly straightforward.

With multidimensional data, however, visualization is more involved. In addition to univariate and bivariate relationships, variables may have interactions such that the relationship between any two variables changes depending on the remaining variables. Standard one- and two-variable plots do not allow us to look at interactions between

multiple variables, and must therefore be complemented with techniques specifically designed for multidimensional data. In this section, we discuss the following tools:

- **Scatterplot Matrix:** displays an array of pairwise scatter plots illustrating the relationship between any pair of variables.
- **Parallel Plot:** displays the variables in a data set as horizontal panels, and connects the values for a particular observation with a set of line segments.

Scatterplot Matrices

A *scatterplot matrix* is a powerful graphical tool that enables you to quickly visualize multidimensional data. It is an array of pairwise scatter plots illustrating the relationship between any pair of variables in a multivariate data set. Often, when faced with the task of analyzing data, the first step is to become familiar with the data. Generating a scatterplot matrix greatly facilitates this process.

Figure 3.17 is a scatterplot matrix of the variables in `fuel.frame`:

```
> splom(~fuel.frame)
```

Note that the factor variable `Type` has been converted to a numeric variable and plotted. The six levels of `Type` (`Compact`, `Large`, `Medium`, `Small`, `Sporty`, and `Van`) simply take the values 1 through 6 in this conversion.

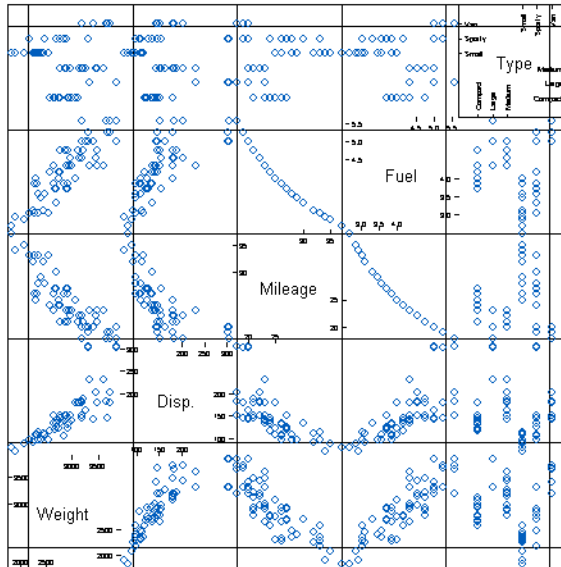


Figure 3.17: Scatterplot matrix of `fuel.frame` using the `splom` function.

Parallel Plots

A *parallel coordinates plot* displays the variables in a data set as horizontal panels, and connects the values for a particular observation with a set of line segments. These kinds of plots show the relative positions of observation values as coordinates on parallel horizontal panels.

Figure 3.18 is a parallel coordinates display of the variables in `fuel.frame`:

```
> parallel(~fuel.frame)
```

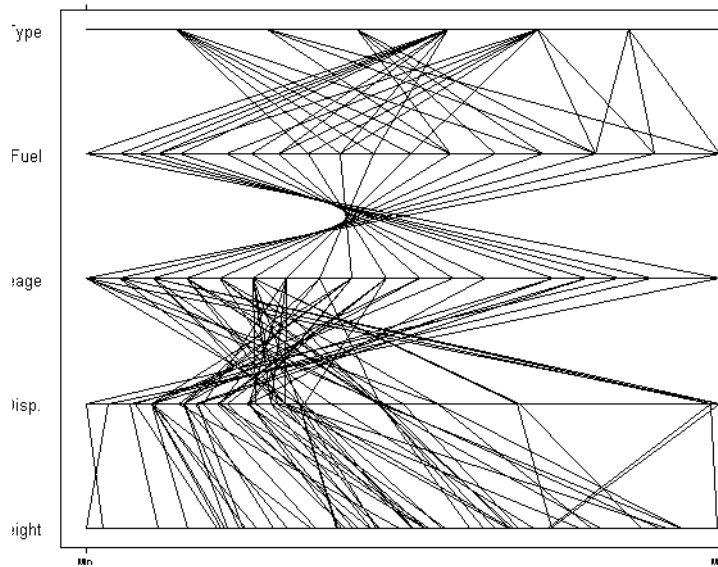


Figure 3.18: *Parallel coordinates plot of the fuel.frame data set using the parallel function.*

Summary: The Display Functions and Their Formulas

The list below organizes the general display functions and their formulas; in doing so, it shows certain conventions and consistencies in the formula mechanism.

Graph one numerical variable against another

```
xyplot(numeric1 ~ numeric2)
```

Graph the sample distribution of one data set

```
densityplot(~numeric)  
histogram(~numeric)  
qqmath(~numeric)
```

Graph measurements with labels

```
barchart(character ~ numeric)  
dotplot(character ~ numeric)  
piechart(character ~ numeric)
```

Compare the sample distributions of two or more data sets

```
bwplot(factor ~ numeric)  
stripplot(factor ~ numeric)  
qq(factor ~ numeric)
```

Graph a function of two variables evaluated on a grid

```
contourplot(numeric1 ~ numeric2*numeric3)  
levelplot(numeric1 ~ numeric2*numeric3)  
wireframe(numeric1 ~ numeric2*numeric3)
```

Graph three numerical variables

```
cloud(numeric1 ~ numeric2*numeric3)
```

Graph multivariate data

```
sploM(~data.frame)  
parallel(~data.frame)
```

ARRANGING SEVERAL GRAPHS ON ONE PAGE

Using the `print` function, you can arrange graphs created separately by Trellis display functions onto a single page. The examples in this section illustrate this feature. Note that these examples do not apply to graphs containing multipanel conditioning that extend for more than one page, such as those created in the section Columns, Rows, and Pages (page 149).

Figure 3.19 shows two graphs arranged on one page:

```
> box.plot <- bwplot(Type~Mileage, data=fuel.frame)
> scatter.plot <- xyplot(Mileage~Weight, data=fuel.frame)
> print(box.plot, position=c(0, 0, 1, 0.4), more=TRUE)
> print(scatter.plot, position=c(0, 0.35, 1, 1))
```

The `position` argument specifies the position of each graph on the page, using a coordinate system in which the lower left corner of the page is 0,0 and the upper right corner is 1,1. The *graph rectangle* is the portion of the page allocated to a graph. The `position` argument takes a vector of four numbers: the first two numbers are the coordinates of the lower left corner of the graph rectangle, and the second two numbers are the coordinates of the upper right corner. If the argument `more=TRUE`, Spotfire S+ expects more drawing on the page with an additional call to `print`.

The following code illustrates the `split` argument to `print`, which provides a different method for arranging plots on a page:

```
> other.plot <- xyplot(Mileage~Disp., data=fuel.frame)
> print(scatter.plot, split=c(1,1,1,2), more=T)
> print(other.plot, split=c(1,2,1,2))
```

The `split` argument accepts a numeric vector of four values. The last two values define an array of subregions on the page; in our example, the array has one column and two rows. The first two values of `split` prescribe the subregion in which the current plot is drawn. In the above code, `scatter.plot` is drawn in the subregion defined by the first column and first row, and `other.plot` is drawn in the subregion defined by the first column and second row.

For more details on the `print` function as it is used in this section, see the help file for `print.trellis`.

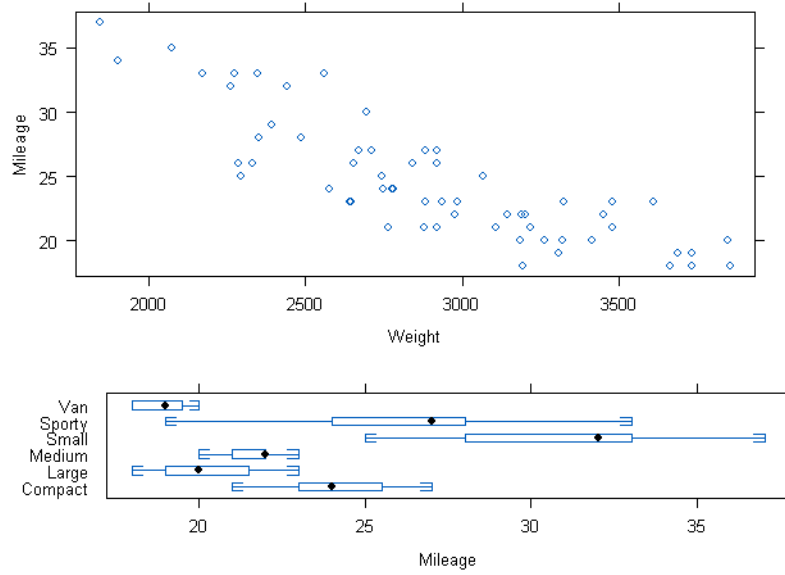


Figure 3.19: Multiple Trellis graphs on a single page using the `print` function.

MULTIPANEL CONDITIONING

About Multipanel Display

Suppose you have a data set based on multiple variables, and you want to see how plots of two variables change in relation to a third “conditioning” variable. With Trellis graphics, you can view your data in a series of panels, where each panel contains a subset of the original data divided into intervals of the conditioning variable. When a conditioning variable is categorical, Spotfire S+ generates plots for each level. When a conditioning variable is numeric, conditioning is automatically carried out on the sorted unique values; each plot represents either an equal number of observations or an equal range of values.

We illustrate the main options for multipanel conditioning using the built-in data set `barley`, which contains observations from a 1930s agricultural field trial that studied barley crops. At six sites in Minnesota, ten varieties of barley were grown for each of two years, 1931 and 1932. The data are the yields for all combinations of site, variety, and year, so there are a total of $6 \times 10 \times 2 = 120$ observations:

```
> names(barley)
[1] "yield" "variety" "year" "site"

> dim(barley)
[1] 120 4
```

The `yield` variable is numeric and all others are factors.

Figure 3.20 uses multipanel conditioning to display the `barley` data. Each panel displays the yields of the ten varieties for one year at one site; `variety` is graphed along the vertical scale and `yield` is graphed along the horizontal scale. For example, the lower left panel displays values of `variety` and `yield` for Grand Rapids in 1932. The *panel variables* are `yield` and `variety`, and the *conditioning variables* are `year` and `site`.

Figure 3.20 is created with the following command:

```
> dotplot(variety ~ yield | year*site, data = barley)
```

The pipe character "|" is read as "given." Thus, you can read the formula as "variety is graphed against yield, given year and site." This simple use of the formula argument creates a complex multipanel display.

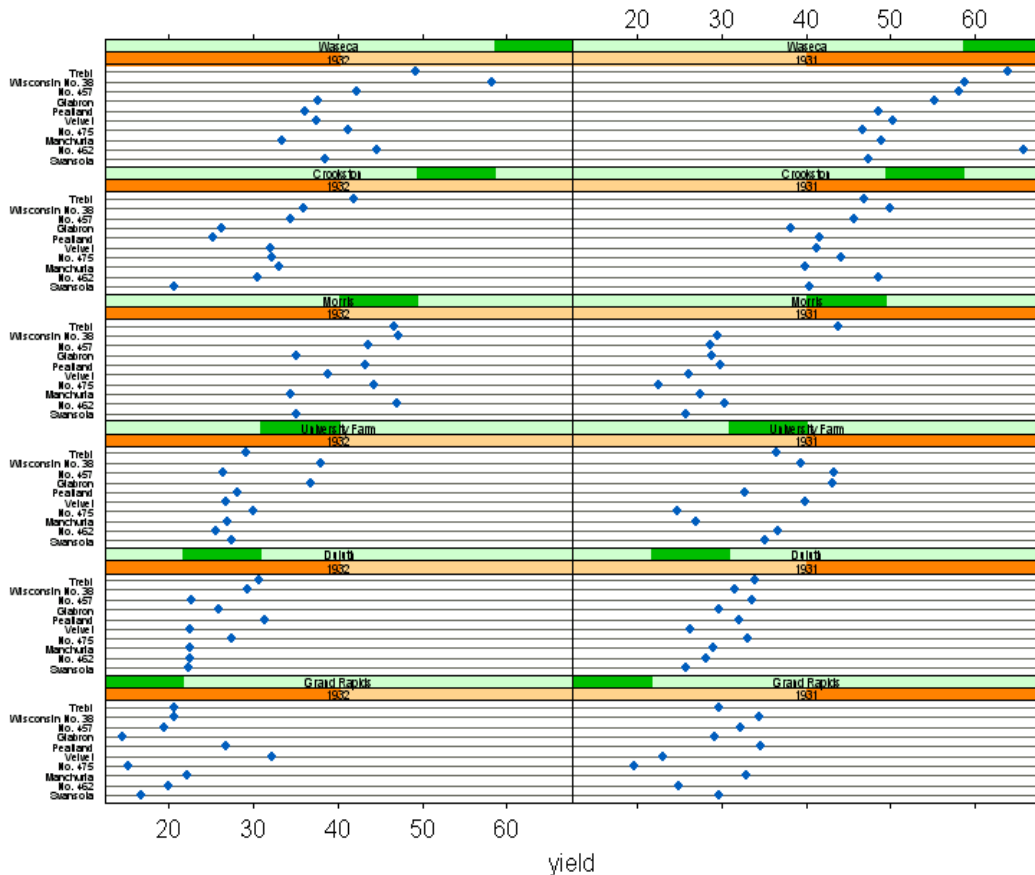


Figure 3.20: Multipanel conditioning for the barley data.

Columns, Rows, and Pages

A multipanel conditioning display is a three-way rectangular array of columns, rows, and pages. For example, there are two columns, six rows, and one page in Figure 3.20. The dimensions of a multipanel array are selected by an algorithm that attempts to fill as much of the graphics region as possible, subject to certain constraints. The constraints include the aspect ratio, the number of conditioning variables, and the number of levels of each conditioning variable.

You can override the dimensions chosen by the layout algorithm by specifying the `layout` argument explicitly. For example, `variety` (an explanatory variable) appears as a panel variable in Figure 3.20. With the command below, we create a new display with `site` as a panel variable instead:

```
> dotplot(site ~ yield | year*variety, data = barley,
+ layout = c(2,5,2))
```

The `layout` argument accepts a numeric vector that specifies the numbers of columns, rows, and pages, respectively. Thus, Figure 3.21 shows the first page of plots in the result, consisting of two columns and five rows. Figure 3.22 shows the second page of plots, also consisting of two columns and five rows.

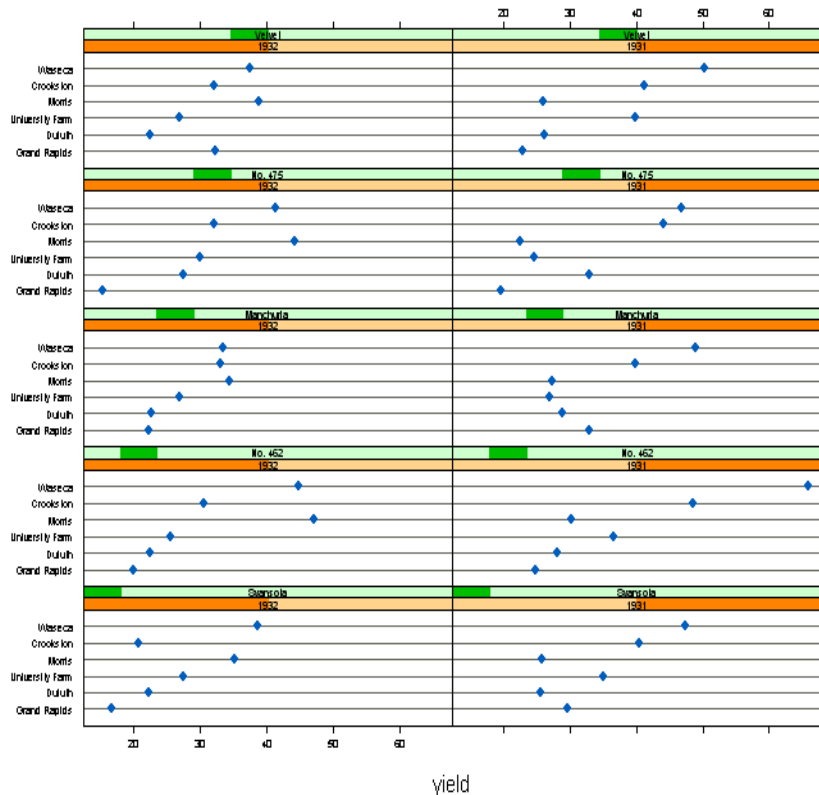


Figure 3.21: The first page of plots for the `barley` data.

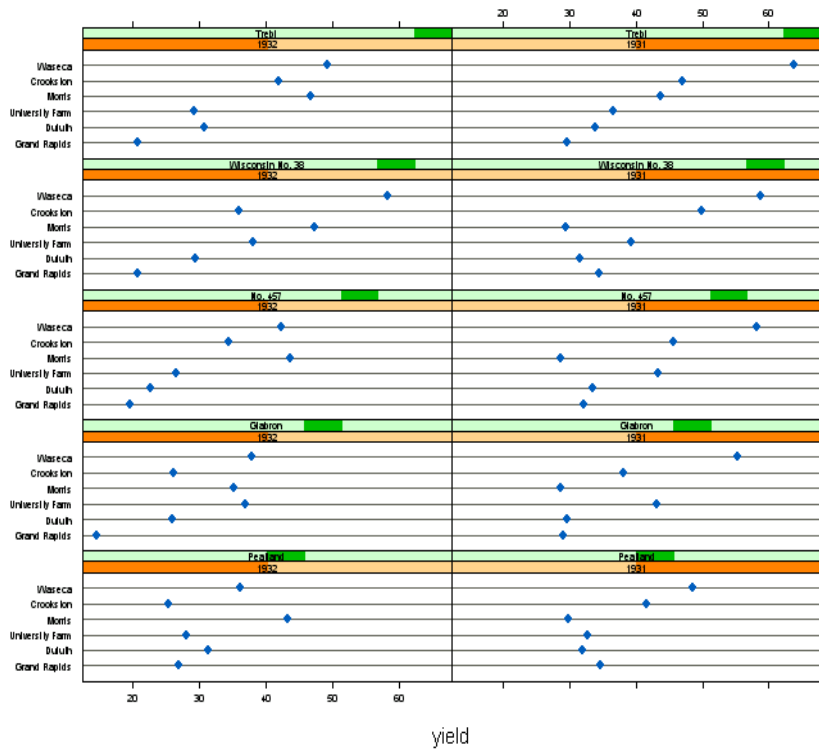


Figure 3.22: The second page of plots for the barley data.

Packet Order and Panel Order

In a multipanel display, a *packet* is the information sent to a panel for a particular plot. In Figure 3.20, each packet includes the values of variety and yield for a particular combination of year and site. Packets are ordered by the levels attribute of the conditioning variables; the levels of the first conditioning variable vary the fastest, the levels of the second conditioning variable vary the next fastest, and so forth. The order of the conditioning variables themselves is determined by the formula used to create a Trellis graph.

As an example, consider the command that generates Figure 3.20, which we reproduce here for convenience:

```
> dotplot(variety ~ yield | year*site, data = barley)
```

The conditioning variable `year` appears first in the formula and `site` appears second. This means that the levels of `year` vary the fastest from packet to packet. In particular, the order of the packets is:

```
1932 Grand Rapids
1931 Grand Rapids
1932 Duluth
1931 Duluth
1932 University Farm
1931 University Farm
1932 Morris
1931 Morris
1932 Crookston
1931 Crookston
1932 Waseca
1931 Waseca
```

The year 1932 is first in the packet ordering because of the `levels` attribute of the `year` variable. Likewise, Grand Rapids appears first because of the `levels` attribute of the `site` variable:

```
> levels(barley$year)
[1] "1932" "1931"

> levels(barley$site)
[1] "Grand Rapids" "Duluth" "University Farm"
[4] "Morris" "Crookston" "Waseca"
```

The panels in a multipanel display are also ordered, from left to right and from bottom to top. The bottom left panel is panel 1. From there, we move fastest through the columns, next fastest through the rows, and slowest through the pages. The panel ordering corresponds to graphs and not to tables: the origin is in the lower left corner, and as we move either from left to right or from bottom to top, the panel order increases.

The following is the panel order for Figure 3.20:

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

In Trellis Graphics, packets are assigned to panels according to both the packet order and the panel order. The information in packet 1 is drawn in panel 1, the information in packet 2 is drawn in panel 2, and so forth. In Figure 3.20, the two orderings result in the year variable changing along the columns and the site variable changing along the rows. Thus, data for 1932 appear in the panels on the left side of the figure, data for the Grand Rapids site appear at the bottom of the figure, and data for the Waseca site appear at the top of the figure. Note that as the levels for a factors change, the darkened bars in the strip labels move from left to right.

Main-Effects Ordering

The plots in Figure 3.20, Figure 3.21, and Figure 3.22 use an important display method called *main-effects ordering of levels*. This method displays the levels of a categorical variable according to some function of the response variable, such as the median. Main effects ordering greatly enhances our ability to perceive effects. In fact, it is so important in Trellis Graphics that Spotfire S+ includes a function `reorder.factor` designed specifically for it; we discuss this function in more detail below.

For the `barley` data, each of the four explanatory variables are factors and the response variable `yield` is numeric. Consider the median yield for each level of the factor variables. We can compute the medians for `variety` with the following command:

```
> variety.medians <- tapply(
+   barley$yield, barley$variety, FUN=median)
> variety.medians
```

Svansota	No. 462	Manchuria	No. 475	Velvet	Peatland
28.55	30.45	30.96667	31.06667	32.15	32.38334
Glabron	No. 457	Wisconsin	No. 38	Trebi	
32.4	33.96666		36.95	39.2	

Notice that the order of the levels in `variety.medians` is the same as the order returned by the `levels` attribute:

```
> levels(barley$variety)
[1] "Svansota"      "No. 462"      "Manchuria"
[4] "No. 475"      "Velvet"       "Peatland"
[7] "Glabron"      "No. 457"      "Wisconsin No. 38"
[10] "Trebi"
```

This is not a coincidence. The levels of `variety` have been specifically sorted in order of increasing `yield` medians. Therefore, Spottfire S+ recognizes `variety` as an *ordered factor* instead of simply a factor:

```
> data.class(barley$variety)
[1] "ordered"
```

This is also true of the other categorical variables in the `barley` data set. As a result, the varieties in Figure 3.20 are ordered in each panel by the `yield` medians: Svansota has the smallest median and appears at the bottom of each panel, and Trebi has the largest median and appears at the top of each panel. Likewise, the panels are ordered by the `yield` medians for `site`: Grand Rapids has the smallest median and appears at the bottom of the figure, and Waseca has the largest median and appears at the top. Finally, the panels are also ordered from left to right by the `yield` medians for `year`: 1932 has the smaller median and 1931 has the larger.

Main-effects ordering is achieved by making each explanatory variable an ordered factor, where the levels are ordered by the medians of the response variable. For example, suppose `variety` is a factor without the median ordering. We can obtain the ordered factor with the following command:

```
# First assign barley to your working directory.
> barley <- barley
> barley$variety <- ordered(barley$variety,
+   levels = names(sort(variety.medians)))
```

To simplify this process, Trellis Graphics includes a function named `reorder.factor` that reorders the levels of a factor variable. Here, it is used to reorder `variety` according to the medians of `yield`:

```
> barley$variety <- reorder.factor(
+   barley$variety, barley$yield, median)
```

The first argument to `reorder.factor` is the factor to be reordered, and the second is the data vector on which the main-effects ordering is based. The third argument is the function to be applied to the second argument to compute main effects.

Conditioning on the Values of a Numeric Variable

In the examples presented so far in this section, we have used the `barley` data set, in which all of the conditioning variables are factors. It is also possible to condition Trellis graphs on the values of a numeric variable. If there are only a few unique values in a numeric variable, we might want to condition plots on the individual values. This produces a display identical to the one we would see if we coerce the variable to class "factor". If there are too many unique values, however, we must condition plots on intervals of the numeric variable. We discuss these two options in detail below.

In the examples that follow, we use the built-in `ethanol` data set, which contains three variables from an industrial experiment with 88 runs:

```
> names(ethanol)
[1] "NOx" "C" "E"

> dim(ethanol)
[1] 88 3
```

The concentrations of oxides of nitrogen (NOx) in the exhaust of an engine were measured for different settings of compression ratio (C) and equivalence ratio (E). These measurements are part of the same experiment that produced the gas data set introduced in the section Giving Data to Trellis Functions (page 120).

Conditioning on Unique Values

In this example, we examine the relationships in `ethanol` between NOx and E for various values of C. The conditioning variable C is numeric and has 5 unique values: 7.5, 9.0, 12.0, 15.0, and 18.0. The `table` function displays the number of observations that correspond to each of these values:

```
> table(ethanol$C)

 7.5  9  12  15  18
 22 17  14  19  16
```

We create scatter plots of NOx versus E for each of the unique values in C:

```
> xyplot(NOx ~ E|C, data=ethanol, layout=c(1,5,1),
+       aspect=1/2)
```

The result is shown in Figure 3.23. When the unique values of a numeric variable are used to condition a Trellis graph, Spotfire S+ determines the packet order by sorting the values. The individual scatter plots are therefore placed in order from $C=7.5$ to $C=18$ in Figure 3.23: the plot for $C=7.5$ appears in the lower left corner of the figure and the plot for $C=18$ appears at the top, as indicated by the darkened bars in the strip labels. For more details on packet order in multipanel displays, see the section Packet Order and Panel Order (page 151).

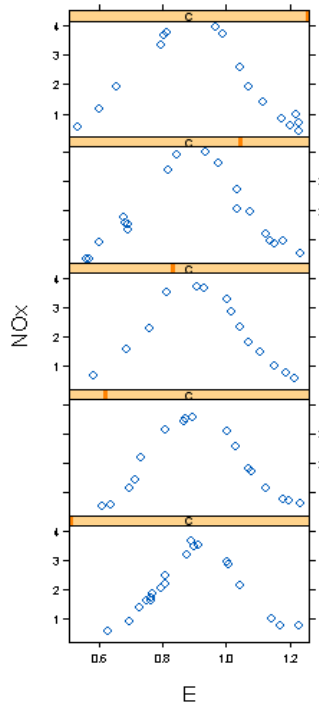


Figure 3.23: *Multipanel conditioning for the ethanol data, using C as the conditioning variable.*

Conditioning on Intervals

In the examples below, we examine the relationships in ethanol between NOx and C for various values of E. The numeric variable E varies in a nearly continuous way: there are 83 unique values out of 88 observations. Clearly we cannot condition on unique values, so we condition on intervals instead. In this type of situation, each panel represents either an equal number of observations or an equal range

of values. We use the `equal.count` function to condition on intervals containing equal numbers of observations, and we use the `shingle` function to condition on intervals containing equal ranges of values.

The `equal.count` function

The Spotfire S+ function `equal.count` implements the *equal count algorithm*, which bins a numeric vector into intervals. The algorithm selects interval endpoints that are values of the data: the left endpoint of the lowest interval is the minimum of the data, and the right endpoint of the highest interval is the maximum of the data. The endpoints are chosen so that the number of observations across intervals is nearly equal, while the fraction of points shared by successive intervals is as close as possible to a specified target fraction.

For example, we can bin the E values in the `ethanol` data with the following command:

```
> GIVEN.E <- equal.count(ethanol$E, number=9, overlap=1/4)
```

The `number` argument specifies the number of intervals, and `overlap` specifies the target fraction of points to be shared by each pair of successive intervals. In the above command, `equal.count` attempts to bin E into nine intervals, while keeping the fraction of points shared by successive intervals as close to 1/4 as possible:

```
> GIVEN.E
```

```
Data:
```

```
[1] 0.907 0.761 1.108 1.016 1.189 1.001 1.231 1.123 1.042
[10] 1.215 0.930 1.152 1.138 0.601 0.696 0.686 1.072 1.074
[19] 0.934 0.808 1.071 1.009 1.142 1.229 1.175 0.568 0.977
[28] 0.767 1.006 0.893 1.152 0.693 1.232 1.036 1.125 1.081
[37] 0.868 0.762 1.144 1.045 0.797 1.115 1.070 1.219 0.637
[46] 0.733 0.715 0.872 0.765 0.878 0.811 0.676 1.045 0.968
[55] 0.846 0.684 0.729 0.911 0.808 1.168 0.749 0.892 1.002
[64] 0.812 1.230 0.804 0.813 1.002 0.696 1.199 1.030 0.602
[73] 0.694 0.816 1.037 1.181 0.899 1.227 1.180 0.795 0.990
[82] 1.201 0.629 0.608 0.584 0.562 0.535 0.655
```

```
Intervals:
```

```
   min   max count
0.535 0.686    13
0.655 0.761    13
```

```
0.733 0.811 12
0.808 0.899 13
0.892 1.002 13
0.990 1.045 13
1.042 1.125 12
1.115 1.189 13
1.175 1.232 13
```

```
Overlap between adjacent intervals:
[1] 4 3 3 3 4 3 3 4
```

With the following command, we use `GIVEN.E` to produce the Trellis graph shown in Figure 3.24:

```
> xyplot(NOx ~ C|GIVEN.E, data = ethanol, aspect = 2.5)
```

The automatic layout algorithm chooses five columns and two rows to display the nine panels. The intervals, which are portrayed by the darkened bars in the strip labels, are ordered from low to high. As we go from left to right and from bottom to top through the panels, the data values in the intervals increase.

In Figure 3.24, the aspect ratio is chosen as 2.5 to bank the underlying pattern of points to approximately 45° . For more information on aspect ratio, see the section Aspect Ratio (page 189).

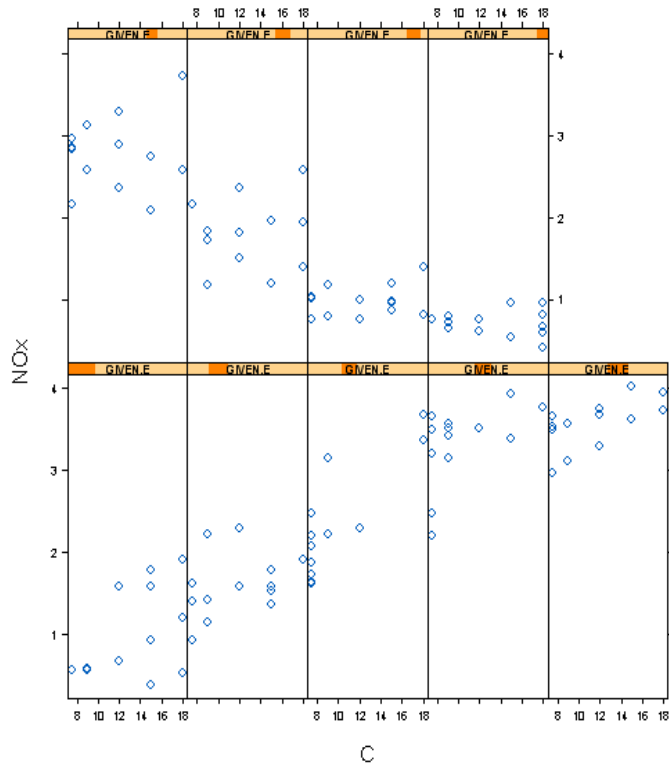


Figure 3.24: *Multipanel conditioning for the ethanol data, using E as the conditioning variable. The equal.count function is used to bin the values in E.*

The shingle function

The result of a call to `equal.count` is an object of class "shingle". The class is named `shingle` because the overlap in the intervals is similar to shingles on a roof. An object of class "shingle" contains the original numerical values in addition to information about the intervals. It can therefore be treated as an ordinary vector. For example, the `range` function works for a `shingle` object just as it does for a vector:

```
> range(GIVEN.E)
[1] 0.535 1.232
```

The endpoints of the intervals are attached to `shingle` object as an attribute. You can use the `levels` function to extract the intervals:

```
> levels(GIVEN.E)
      min  max
0.535 0.686
0.655 0.761
0.733 0.811
0.808 0.899
0.892 1.002
0.990 1.045
1.042 1.125
1.115 1.189
1.175 1.232
```

Because of this, you can graphically display the intervals with a special `plot` method. Figure 3.25 shows the intervals in `GIVEN.E`:

```
> plot(GIVEN.E)
```

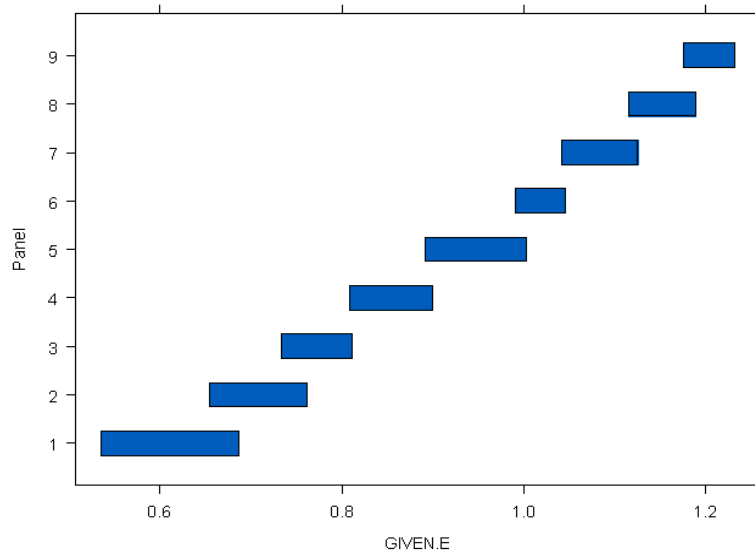


Figure 3.25: Plot of `GIVEN.E`, an object of class "shingle".

You can create an object of class "shingle" directly with the function `shingle`. This is most useful for computing intervals based on an algorithm that is different than the `equal.counts` function. For

example, the following commands create five intervals of equal width from the E column in `ethanol`. The intervals include no overlapping points:

```
> endpoints <- seq(
+   from=min(ethanol$E), to=max(ethanol$E), length=6)

> GIVEN.E2 <- shingle(ethanol$E,
+   intervals = cbind(endpoints[-6], endpoints[-1]))
```

The `intervals` argument is a two-column matrix holding the left and right endpoints of the intervals. The width of each resulting interval is approximately 0.14:

```
> levels(GIVEN.E2)

      min      max
0.5350 0.6744
0.6744 0.8138
0.8138 0.9532
0.9532 1.0926
1.0926 1.2320
```

With the following command, we can use `GIVEN.E2` to produce a Trellis graph conditioned on these intervals:

```
> xyplot(NOx ~ C|GIVEN.E2, data = ethanol, aspect = 2.5)
```

Summary: The Layout of a Multipanel Display

The following aspects control the layout of a multipanel display:

- The order of the conditioning variables in the formula argument determines the packet order. The levels, values, or intervals of the first conditioning variable vary the quickest in the packet order; those of the last conditioning variable vary the slowest.
- For categorical conditioning variables, the `ordered` and `reorder.factor` functions can be used to control the levels in the packet order. For numeric conditioning variables, the values in the packets are automatically sorted in increasing order.
- The number of columns, rows, and pages in the multipanel display is determined by the `layout` argument. If `layout` is not specified, a default algorithm is used.

GENERAL OPTIONS FOR MULTIPANEL DISPLAYS

Spacing Between Rows and Columns

You can use the `general` argument between to insert space between adjacent rows or adjacent columns of a multipanel Trellis display. To illustrate this argument, we use the built-in data set `barley` introduced in the section About Multipanel Display (page 148).

The following commands display the `barley` data in a way similar to that shown in Figure 3.21 and Figure 3.22. In the resulting two-page graphic, `yield` is plotted against `site` given `variety` and `year`:

```
> barley.plot <- dotplot(site ~ yield | variety*year,  
+   data=barley, aspect="xy", layout=c(2,5,2))  
  
> print(barley.plot)
```

The defined layout places the measurements for 1931 on the first page and those for 1932 on the second page. We can squeeze the panels onto one page by changing the `layout` argument to `(2, 10, 1)`. To do this, we update the `barley.plot` object:

```
> barley.plot <- update(barley.plot, layout=c(2,10,1))  
> print(barley.plot)
```

The result is shown in Figure 3.26. Rows 1 through 5 from the bottom of the figure show the 1932 data, and rows 6 through 10 show the 1931 data.

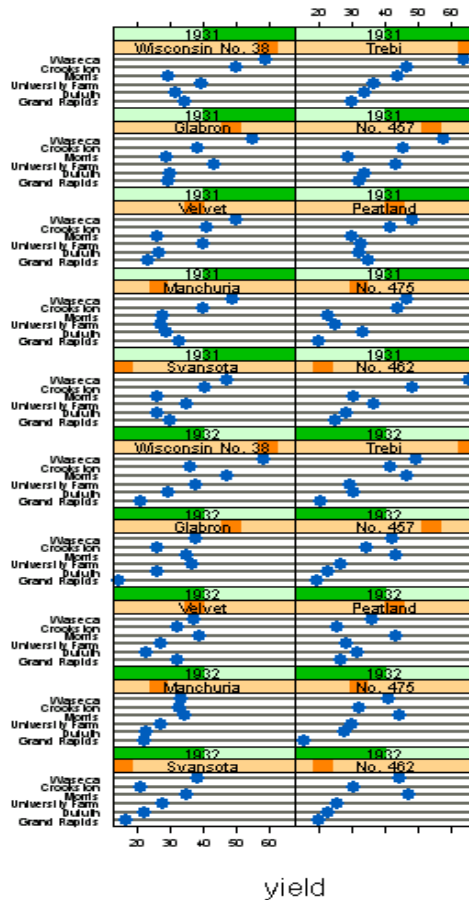


Figure 3.26: Multipanel dot plot of the barley data. Rows 1 through 5 show the 1932 data, and rows 6 through 10 show the 1931 data.

In the figure, the change in value of the year variable is indicated by the text on the strip labels. We can provide a stronger indication of the change by adding space in the display between rows 5 and 6, so that a break occurs between the 1932 panels and the 1931 panels. For this purpose, we use the `between` argument, which accepts a list with components `x` and `y`. The `x` component is a vector with length equal to one less than the number of columns in the multipanel display. The values in `x` give the amount of space to be inserted between each pair of adjacent columns, measured in units of character height. Similarly,

the `y` component specifies the amount of space between each pair of adjacent rows. Either `x` or `y` can be missing, indicating that no space should be added.

For example, the following command adds space between rows 5 and 6 of the display shown in Figure 3.26:

```
> barley.plot <- update(barley.plot,  
+   between = list(y=c(0,0,0,0,1,0,0,0,0)))  
> print(barley.plot)
```

Skipping Panels

You can use the general argument `skip` to skip particular panels in a multipanel display. The `skip` argument accepts a logical vector that contains as many values as there are panels in one page of the display. Each element of `skip` indicates whether to skip the corresponding panel.

To illustrate this argument, we use the built-in data set `market.survey`, which contains 10 columns of demographic data compiled from an AT&T telephone survey. The following commands display box plots of the `market.survey` data conditioned on the two variables `income` and `pick`:

```
> market.plot <- bwplot(age ~ log(1+usage) | income*pick,  
+   strip = function(...)  
+     strip.default(..., strip.names=c(T,T)),  
+   skip = c(F,F,F,F,F,F,F,T),  
+   layout = c(2,4,2),  
+   data = market.survey)  
  
> print(market.plot)
```

The chosen layout has two pages, each containing eight panels and seven plots. On each page, the last panel is skipped because the conditioning variable `income` has only seven levels.

For more details about the `strip` argument as it is used in this example, see the section [Changing the Text in Strip Labels](#) (page 168).

Multipage Displays

You can use the general argument `page` to add page numbers, text, or graphics to each page of a multipage Trellis display. The `page` argument is a function that accepts a single argument `n`, the page

number, and issues drawing commands specific to page *n*. For example, the following updates the `market.plot` object from the previous section so that it includes page numbers:

```
> update(market.plot, page = function(n)
+   text(x=0.75, y=0.95, paste("page",n), adj=0.5))
```

The `text` function is a Spotfire S+ traditional graphics command that uses a coordinate system in which 0,0 is the lower left corner of the page and 1,1 is the upper right corner.

Hint

If a multipage display is sent to a screen device, the default behavior draws each page in order without pausing between pages. You can force the screen device to prompt you before drawing each page by typing

```
> par(ask=TRUE)
```

before issuing your graphics commands.

SCALES AND LABELS

All of the functions presented in the section General Display Functions (page 124) have arguments that specify the scales and labels of graphs. We discuss these arguments in detail here.

Axis Labels and Titles

The following command displays a scatter plot of NO_x against E for the gas data set, which was introduced in the section Giving Data to Trellis Functions (page 120):

```
> xyplot(NOx~E, data=gas, aspect=1/2)
```

The default axis labels are the names of the variables used in the formula argument. We can specify more descriptive axis labels, as well as a main title and a subtitle, using the following command:

```
> xyplot(NOx~E, data=gas, aspect=1/2,
+   xlab = "Equivalence Ratio",
+   ylab = "Oxides of Nitrogen",
+   main = "Air Pollution",
+   sub = "Single-Cylinder Engine")
```

The result is shown in Figure 3.27. Note that the main title appears at the top of the graph, and the subtitle appears at the bottom of the graph under the horizontal axis label.

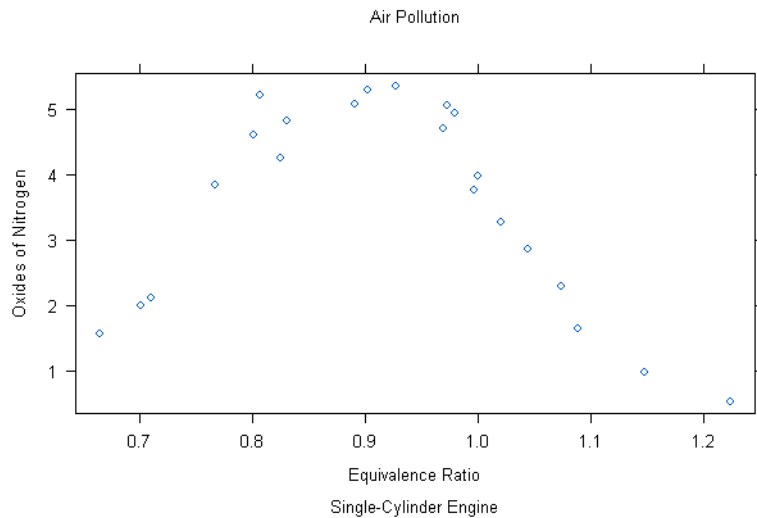


Figure 3.27: Scatter plot of the gas data that includes titles and axis labels.

Each of the four label arguments `xlab`, `ylab`, `main`, and `sub` can be a list. When specified as a list, the first component is a character string for the text of the label. The remaining components specify the size, font, and color of the text in the label. The component `cex` specifies the size, `font` is a positive integer that specifies the font type, and `col` is a positive integer that specifies the color. For example, the following code changes the sizes of the title and subtitle in Figure 3.27:

```
> xyplot(N0x~E, data=gas, aspect=1/2,
+   xlab = "Equivalence Ratio",
+   ylab = "Oxides of Nitrogen",
+   main = list("Air Pollution", cex=2),
+   sub = list("Single-Cylinder Engine", cex=1.25))
```

Axis Limits

In Trellis graphics, the upper axis limit for a numeric variable is the maximum of the data to be plotted plus 4% of the range of the data. Similarly, the lower axis limit is the minimum of the data minus 4% of the range of the data. The extra 4% at each end of the data values prevents the extreme values from being plotted on edges of the plot.

We can alter the default limits with the arguments `xlim` and `ylim`. The `xlim` argument is a numeric vector of two values specifying the minimum and maximum limits on the horizontal axis. Similarly, `ylim` is a vector of two values specifying the minimum and maximum limits on the vertical axis. For example, the range of the `N0x` variable in the `gas` data set is:

```
> range(gas$N0x)
[1] 0.537 5.344
```

The following command specifies the values 0 and 6 as the limits of the vertical axis in a scatter plot of `gas`:

```
> xyplot(N0x~E, data=gas, aspect=1/2, ylim=c(0,6))
```

Tick Marks and Labels

The general argument `scales` affects tick marks and tick labels in Trellis graphics. With `scales`, you can change both the number of ticks and the size of the tick labels. For example, the `xyplot` command above results in seven ticks on the vertical axis and six on the horizontal axis. With the following command, we reduce the number of ticks and increase the size of the tick labels:

```
> xyplot(N0x~E, data=gas, aspect=1/2, ylim = c(0,6),
+   scales = list(cex=2, tick.number=4))
```

The argument `scales` accepts a list with two components: the `cex` component affects the size of the tick labels and the `tick.number` component affects the number of ticks. Note that Spotfire S+ interprets `tick.number` as a suggestion only. An algorithm finds a set “nice” tick values that is as close in number to `tick.number` as possible.

We can also specify the tick marks and labels separately for each axis. For example, the specification

```
scales = list(cex=2,  
             x = list(tick.number=4),  
             y = list(tick.number=10))
```

changes `cex` on axes. However, `tick.number` is 4 for the horizontal axis and 10 for the vertical axis. Thus, specifications for the horizontal axis only appear in the argument `scales` as a component `x` that is itself a list, specifications for the vertical axis only appear in `scales` as a component `y` that is a list, and specifications for both axes appear as remaining components of the argument `scales`.

The `scales` argument can contain many different list components in addition to `cex` and `tick.number`. For more details, see the online help file for `trellis.args`.

Exceptions

The two general display functions `wireframe` and `cloud` currently do not accept changes to each axis separately. Thus, components `x`, `y`, and `z` cannot be used in the `scales` argument.

The general display function `piechart` has no tick marks or tick labels, so the `scales` argument does not apply at all.

The general display function `splom` has many scales, so only limited control over the axes is available through the argument `scales`.

Changing the Text in Strip Labels

The default text in the strip labels of a multipanel display is derived from the names and levels of the conditioning variables. If a conditioning variable is categorical, the strip label for each panel is the name of the corresponding factor level. The `barley` data set introduced in the section *About Multipanel Display* (page 148) illustrates this:

```
> dotplot(variety ~ yield | year*site, data = barley)
```

The strip labels in the resulting graphic contain the levels of the year and site variables.

If a conditioning variable is numeric, however, the strip labels for all panels simply contain the name of the variable. This is illustrated with the ethanol data introduced in the section Conditioning on the Values of a Numeric Variable (page 155):

```
> xyplot(NOx ~ E|C, data = ethanol)
```

The strip label “C” appears in all five panels of the resulting graphic.

One way to change the default strip labels is to change the names of the factor levels or numeric variables directly. For example, suppose we want to change the long label “University Farm” to “U. Farm” in conditioned plots of the barley data. We can change the names of the levels of the site variable as follows:

```
> levels(barley$site)

[1] "Grand Rapids" "Duluth" "University Farm"
[4] "Morris" "Crookston" "Waseca"

# First assign barley to your working directory.
> barley <- barley
> levels(barley$site)[3] <- "U. Farm"

> levels(barley$site)

[1] "Grand Rapids" "Duluth" "U. Farm"
[4] "Morris" "Crookston" "Waseca"
```

The general arguments `par.strip.text` and `strip` provide additional control over the look of strip labels in conditioned Trellis graphs.

The `par.strip.text` argument

The size, font, and color of the text in strip labels can be changed with the argument `par.strip.text`. This argument is a list with the following components: `cex` determines the font size, `font` determines the font type, and `col` determines the text color. For example, we can specify huge strip labels on a plot of the ethanol data with the following command:

```
> xyplot(NOx ~ E|C, data = ethanol,
```

```
+ par.strip.text = list(cex=2))
```

The strip argument

The `strip` argument allows delicate control over the text that is placed in the strip labels. One potential use removes the strip labels altogether:

```
strip = F
```

Another possible use controls the information included in the text of the strip labels. For example, the command below uses the names of the conditioning variables along with their levels in the strip labels:

```
> dotplot(variety ~ yield | year*site, data = barley,  
+ strip = function(...)  
+ strip.default(..., strip.names = c(T,T)))
```

The argument `strip.names` in the `strip.default` function accepts a logical vector of length two. The first element indicates whether the names of factors should be included in strip labels along with the names of factor levels. The second element indicates whether the names of shingles should be included. The default value is `strip.names=c(F,T)`.

The `strip.default` function includes many other arguments that can be modified through `strip` with syntax analogous to the above command. For more details, see the help file for `strip.default`.

PANEL FUNCTIONS

The *data region* of a panel in a Trellis graph is a rectangle that just encloses the data. Panel functions have sole responsibility for drawing in data regions; they are specified by a `panel` argument to the general display functions. Panel functions manage the symbols, lines, and so forth that encode the data in the data regions. The other arguments to the general display functions manage the superstructure of the graph, such as scales, labels, boxes around the data region, and keys.

Every general display function has a default panel function. The name of the default panel function for a particular type of plot is “panel,” followed by a period and the name of the display function. For example, the default panel function for `xypLOT` is `panel.xypLOT`. In all the examples so far in this chapter, default panel functions have been used to draw all of the plots.

You can modify what is drawn in the data region of a plot by one of three mechanisms:

- Pass new values to arguments in a default panel function.
- Write your own custom panel function.
- Modify a special-purpose panel function included in the Trellis library.

In this section, we discuss all three of these options.

Passing Arguments to a Default Panel Function

You can give an argument to a default panel function by passing it to the corresponding general display function. The general display functions automatically pass unrecognized arguments on to the default panel functions. For example, the `panel.xypLOT` function accepts a `pch` argument that controls the plotting character in a scatter plot. We can specify a plus sign for the plotting character by including the argument `pch="+"` in our call to `xypLOT`. The following command does this for a plot of the built-in `gas` data set:

```
> xypLOT(N0x~E, data=gas, aspect=1/2, pch="+")
```

For more details about the arguments accepted by a default panel function, see its online help file.

Writing a Custom Panel Function

Panel functions can accept any number of arguments, but the first two should always be named `x` and `y`. These two arguments represent vectors containing the horizontal and vertical coordinates, respectively, of the points to be displayed in the panels. The remaining arguments can be parameters specific to the display you want to create, traditional graphics parameters, etc.

As an example of a custom panel function, consider the `gas` data set. Suppose you want to use `xyplot` to graph the `N0x` variable against `E`, using “+” as the plotting symbol for all observations except those for which `N0x` is a maximum, in which case you want to use “M.” There is no provision in `xyplot` to do this, so you must write your own panel function. The following command defines a `panel.special` function that accomplishes this:

```
> panel.special <- function(x,y) {
+   biggest <- y==max(y)
+   points(x[!biggest], y[!biggest], pch="+")
+   points(x[biggest], y[biggest], pch="M")
+ }
```

The function `points` is a traditional graphics function that draws individual points in a plot. The first argument to `points` contains the horizontal coordinates of the points to be plotted, the second argument contains the vertical coordinates, and the argument `pch` determines the symbol used to display the points.

We can use the `panel.special` function to plot the `gas` data by passing it to the `panel` argument in our call to `xyplot`. For example:

```
> xyplot(N0x~E, data=gas, aspect=1/2, panel=panel.special)
```

The result is shown in Figure 3.28. A custom panel function can also be defined directly in a call to a general display function. For example, the following command produces the same graphic as the one shown in Figure 3.28:

```
> xyplot(N0x~E, data=gas, aspect=1/2,
+   panel = function(x,y) {
+     biggest <- y==max(y)
+     points(x[!biggest], y[!biggest], pch="+")
+     points(x[biggest], y[biggest], pch="M")
+   }
+ )
```

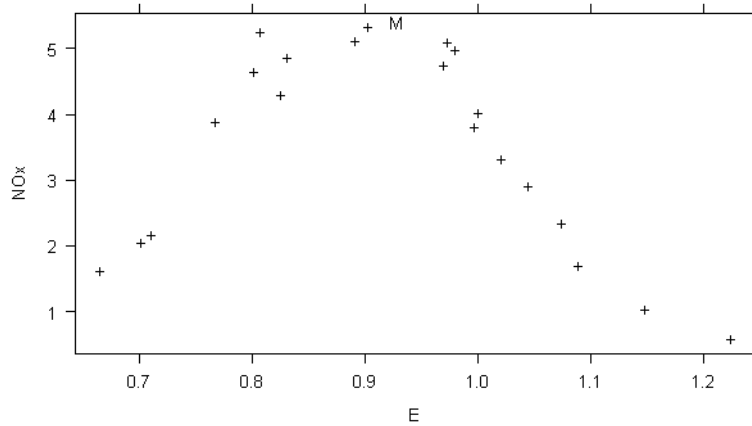


Figure 3.28: Scatter plot of the gas data using the `panel.special` panel function.

In most cases, a panel function that is used for a single-panel display can be used for a multipanel display as well. For example, we can use `panel.special` to show the maximum value of `NOx` on each panel in a display of the `ethanol` data:

```
> xyplot(NOx ~ E|C, data=ethanol, aspect=1/2,
+       panel=panel.special)
```

Special Panel Functions

When writing a custom panel function, you may want to incorporate code from a default panel function as part of it. This is often true when you want to simply augment a standard Trellis panel, without creating a new one from scratch. In addition, the Trellis library provides some special-purpose panel functions that are not attached to particular display functions. One such function is `panel.loess`, which adds smooth curves to scatter plots. Functions such as `panel.loess` are very helpful for quickly augmenting standard panel functions.

For example, to add smooth curves to a multipanel display of the `ethanol` data, type the following:

```
> GIVEN.E <- equal.count(ethanol$E, number=9,
+       overlap=1/4)
```

```
> xyplot(NOx ~ C|GIVEN.E, data=ethanol, aspect=2.5,
+   panel = function(x,y) {
+     panel.xyplot(x,y)
+     panel.loess(x, y, span=1)
+   }
+ )
```

The default panel function `panel.xyplot` draws the data points in each scatter plot. The special panel function `panel.loess` computes and draws the smooth curves. The argument `span` to `panel.loess` is the smoothing parameter; larger values of `span` result in smoother curves and smaller values result in rougher curves. For more details on the `equal.count` function as it is used in this example, see the section [Conditioning on Intervals](#) (page 156).

A particularly useful argument to many default panel functions is `subscripts`. If you request this argument in a custom panel function, the subscripts of the original observations that are included in each panel can be obtained. Knowing these subscripts is helpful for extracting the values of other variables that might be needed to render each panel. In such a case, the panel function argument `subscripts` contains the subscripts. For example, the following command uses `subscripts` to draw the observation numbers on a conditioned plot of the `ethanol` data:

```
> xyplot(NOx ~ E|C, data=ethanol, aspect=1/2,
+   panel = function(x, y, subscripts)
+     text(x, y, subscripts, cex=0.75)
+ )
```

**Summary:
Common
Options in
Panel
Functions**

The traditional graphics functions commonly used in both default and custom panel functions are:

`points`, `lines`, `text`, `segments`, and `polygon`.

Use the Spotfire S+ online help system to see descriptions of each of these functions.

The traditional graphics parameters commonly used in both default and custom panel functions are:

`col`, `lty`, `pch`, `lwd`, and `cex`.

See the online help file for `par` to see descriptions of each of these parameters.

PANEL FUNCTIONS AND THE TRELIS SETTINGS

As we have discussed in this chapter, Trellis Graphics is implemented using the traditional Spotfire S+ graphics system, which has controllable parameters that determine the characteristics of plotted objects. For example, graphical parameters determine the type, size, font, and color of the symbols in a scatter plot. In Trellis Graphics, the default panel functions for the general display functions select graphical parameters to render plotted elements as effectively as possible. Because the most desirable choices for one graphics device can be different from those for another device, the default graphical parameters are device-dependent. These parameters are contained in lists that we refer to as the *Trellis settings*. When `trellis.device` sets up a graphics device, the Trellis settings are established for that device and are saved in a special data structure.

When you write your own custom panel functions, you may want to make use of the Trellis settings to provide good performance across different devices. Three functions enable you to access, display, and change the settings for the current device: `trellis.par.get` allows you to extract settings for use in a panel function, `show.settings` shows the values of the settings graphically, and `trellis.par.set` lets you change the settings for the current device. We discuss each of these functions in this section.

The `trellis.par.get` Function

The `trellis.par.get` function allows you to extract the Trellis settings for particular graphics devices. For example, here is the panel function `panel.xyplot`:

```
> panel.xyplot

function(x, y, type = "p", cex = plot.symbol$cex,
pch = plot.symbol$pch, font = plot.symbol$font,
lwd = plot.line$lwd, lty = plot.line$lty,
col = if(type == "l") plot.line$col else plot.symbol$col,
...)
{
  if(type == "l") {
    plot.line <- trellis.par.get("plot.line")
    lines(x, y, lwd = lwd, lty = lty, col = col,
          type = type, ...)
  }
}
```

```

    }
  else {
    plot.line <- trellis.par.get("plot.line")
    plot.symbol <- trellis.par.get("plot.symbol")
    points(x, y, pch = pch, font = font, cex = cex,
           col = col, type = type, lty = lty, lwd = lwd, ...)
  }
}

```

In this panel function, point symbols are used to plot the data when the argument `type="p"`. The plotting symbol for the points is defined by the settings list `plot.symbol`, which is accessed by `trellis.par.get`. The components of `plot.symbol` are given to the `points` function, which draws the symbols. Here is the `plot.symbol` list for the device (graphsheat for Windows or Motif for UNIX):

```

> trellis.device(devicetype)
##where devicetype is graphsheat or motif##
> plot.symbol <- trellis.par.get("plot.symbol")
> plot.symbol

$cex:
[1] 0.8

$col:
[1] 2

$font:
[1] 1

$pch:
[1] 1

```

The `pch` value of 1 and the `col` value of 2 produces a cyan circle.

In the code for `panel.xyplot`, the `lines` function is used to plot the data when `type="l"`. In this case, the Trellis graphical parameters for lines are extracted from the settings list `plot.line`. For example, here is the `plot.line` list for the device (graphsheat for Windows or Motif for UNIX):

```

> trellis.device(devicetype)
##where devicetype is graphsheat or motif##
> plot.line <- trellis.par.get("plot.line")

```

```
> plot.line  
  
$col:  
[1] 2  
  
$lty:  
[1] 1  
  
$lwd:  
[1] 1
```

These settings produce a cyan-colored solid line.

The `show.settings` Function

The `show.settings` function displays the graphical parameters in the Trellis settings for the current device. For example, to see the settings for the black and white postscript device, type:

```
> trellis.device(postscript, file = "settings.ps")  
> show.settings()  
> dev.off()  
null device  
1
```

This creates a postscript file in your working directory named **settings.ps** that contains a series of plots. Each panel in the file displays one or more lists of settings; the names of the settings appear below the panels. For example, the panel in the third row (from the top) and first column shows plotting symbols determined by `plot.symbol` and lines determined by `plot.line`. To see the value of all the trellis settings for this device, type `trellis.settings.bwps` at the Spotfire S+ prompt.

The `trellis.par.set` Function

The Trellis settings for a particular device can be changed with the `trellis.par.set` function. For example, the following commands change the color of the plotting symbol for the graphsheet (Windows) or Motif (UNIX) device from cyan to magenta:

```
# If on Windows:  
> trellis.device(graphsheet)  
# If on UNIX:  
> trellis.device(motif)
```

```
> plot.symbol <- trellis.par.get("plot.symbol")
> plot.symbol$col
[1] 2

> plot.symbol$col <- 3
> trellis.par.set("plot.symbol", plot.symbol)
> plot.symbol <- trellis.par.get("plot.symbol")
> plot.symbol$col
[1] 3
```

The `trellis.par.set` function sets an entire list of Trellis settings, and not just some of the components. Thus, the simplest way to make a change is to extract the current list with `trellis.par.get`, alter the desired components, and then save the altered list. The change lasts only as long as the device continues. If the Spotfire S+ session is ended or the device is closed, the altered settings are removed.

SUPERPOSING MULTIPLE VALUE GROUPS ON A PANEL

A common visualization technique involves superposing two or more groups of values in the same data region and encoding the groups in different ways. This technique allows you to quickly visualize the similarities and differences between the groups of data values. In Trellis Graphics, superposition is achieved by the special panel function `panel.superpose`. The key argument to the general display functions can be used in conjunction with `panel.superpose` to draw legends that distinguish the points from each group.

Superposing Points

We illustrate the superposition of points with the built-in data set `fuel.frame`, which we introduced in the section `General Display Functions` (page 124). In our examples, we graph the `Mileage` variable against `Weight` for the six types of vehicles described by the factor `Type`.

As a first example, we encode the vehicle types in `fuel.frame` using different plotting symbols. The panel function `panel.superpose` carries out such a superposition:

```
> xyplot(Mileage~Weight, data=fuel.frame, aspect=1,  
+ groups=Type, panel=panel.superpose)
```

The factor `Type` is given to the `groups` argument of `xyplot`, which passes it to the `groups` argument of `panel.superpose`. The `panel.superpose` function then determines the plotting symbol for each group of data points.

The `panel.superpose` function uses the graphical parameters in the Trellis setting `superpose.symbol` for the default plotting symbols. For example, the following lists the settings for the black and white postscript device:

```
> trellis.device(postscript)  
> trellis.par.get("superpose.symbol")  
  
$cex:  
[1] 0.85 0.85 0.85 0.85 0.85 0.85 0.85  
  
$col:  
[1] 1 1 1 1 1 1 1
```

```

$font:
[1] 1 1 1 1 1 1 1

$pch:
[1] "\001" "+" ">" "s" "w" "#" "{"

> dev.off()
null device
1

```

There are seven symbols, so that up to seven groups of data points can be distinguished in a single plot. The symbols are used in sequence: the first two symbols are used if there are two groups, the first three symbols are used if there are three groups, and so on. You can use the `show.settings` function to graphically view the seven symbols for the current device; for more details, see the section [The show.settings Function](#) (page 177).

The default symbols in the `superpose.symbol` list have been chosen to enhance the visual assembly of each group of points. That is, we want to effortlessly assemble the plotting symbols of a given type to form a visual whole. If assembly can be performed efficiently, we can quickly compare the characteristics of the different groups of data.

Using the `pch` argument, you can override the default symbols with your own set of plotting symbols. For example, suppose we want to use the first letters of the vehicle types in our plots of the `fuel.frame` data. First, we define the following character vector to represent our plotting symbols:

```
> mysymbols <- c("C", "L", "M", "P", "S", "V")
```

Here, “C” is for Compact, “L” is for Large, “M” is for Medium, “P” is for Small (to avoid duplication with Sporty), “S” is for Sporty, and “V” is for Van. To use these symbols in a plot, pass `mysymbols` to the `pch` argument in the call to `xypplot`:

```
> xypplot(Mileage~Weight, data=fuel.frame, aspect=1,
+         groups=Type, pch=mysymbols, panel=panel.superpose)
```

The result is shown in [Figure 3.29](#). The `pch` argument passes the vector to `panel.superpose`, which uses it to determine the plotting symbol for each group.

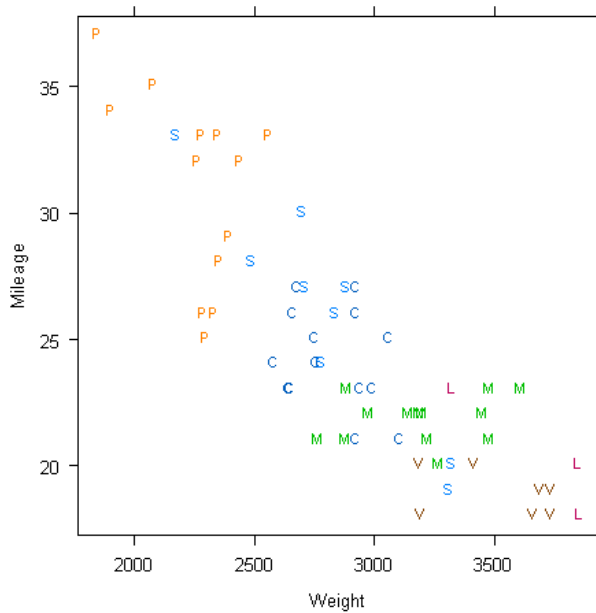


Figure 3.29: Scatter plot of the `fuel.frame` data, using the first letter of each car type for the plotting symbols.

Superposing Curves

The `panel.superpose` function also superposes curves onto the same plot. For example, the following code superposes a line and a quadratic:

```
> x <- seq(from=0, to=1, length=50)
> linquad <- c(x,x^2)
> x <- rep(x, times=2)
> which.curve <- rep(c("linear","quadratic"), c(50,50))
> xyplot(linquad~x, xlab="Argument", ylab="Functions",
+       aspect=1, groups=which.curve, type="l",
+       panel=panel.superpose)
```

The `type="l"` argument specifies that line plots should be rendered. The `panel.superpose` function uses the graphical parameters in the Trellis setting `superpose.line` for the default line types.

To list the settings for the black and white postscript device:

```
> trellis.device(postscript)
> trellis.par.get("superpose.line")

$col:
[1] 1 1 1 1 1 1 1

$lty:
[1] 1 2 3 4 5 6 7

$lwd:
[1] 1 1 1 1 1 1 1

> dev.off()
null device
      1
```

There are seven line types, so that up to seven groups of data points can be distinguished in a single plot. You can use the `show.settings` function to graphically view the seven line types for the current device; for more details, see the section `The show.settings Function` (page 177).

Superposing Other Plots

The function `panel.superpose` can be used with any general display function where superposing different groups of values makes sense. In general, we can superpose data sets using `xyplot`, `dotplot`, or many of the other display functions. For example, the following code produces a dot plot of the barley data introduced in the section `About Multipanel Display` (page 148):

```
> barley.plot <- dotplot(variety ~ yield|site, data=barley,
+   groups=year, layout=c(1,6), aspect=0.5,
+   xlab="Barley Yield (bushels/acre)",
+   panel = function(x,y,...) {
+     dot.line <- trellis.par.get("dot.line")
+     abline(h=unique(y), lwd=dot.line$lwd,
+           lty=dot.line$lty, col=dot.line$col)
+     panel.superpose(x,y,...)
+   }
+ )

> print(barley.plot)
```


On each panel of the resulting figure, data for the years 1931 and 1932 are distinguished by different plotting symbols.

The panel function for `dotplot` is slightly more complicated than the one for `xypplot`, because the horizontal lines of the dot plot must be drawn in addition to the plotting symbols. The `abline` function is used to draw the lines at unique values on the vertical axis; the characteristics of the lines are specified by the Trellis setting `dot.line`. To see the settings for the current graphics device, type `trellis.settings$dot.line` at the Spotfire S+ prompt. For more details, see the help file for `panel.dotplot`.

The key Argument

A key can be added to a Trellis display through the `key` argument of the general display functions. This argument is a list with components that are the names of arguments to the key function, which actually draws the key. Thus, the components in the `key` argument in a general display function are passed directly to the corresponding arguments of the key function. The exception to this is the `space` component, which leaves extra space for a key in the margins of the display; the `space` component does not have a corresponding argument in the key function. For more details, see the help file for `trellis.args`.

A Simple Example

The `key` argument to general display functions is easy to use and yet quite powerful. As a simple example, the following command updates the `barley.plot` object from the previous section:

```
> update(barley.plot, key = list(
+   points=Rows(trellis.par.get("superpose.symbol"),1:2),
+   text=list(levels(barley$year))))
```

The result is shown in Figure 3.30. The `text` component of the `key` argument is a list with the year names. The `points` component is a list with the graphical parameters of the two symbols used to plot the data. We extract these parameters from the Trellis setting `superpose.symbol`, which `panel.superpose` uses to draw the plotting symbols. We want to give the `points` component only the parameters of the symbols used, so we use the function `Rows` to extract the first two elements of each component in `superpose.symbol`. The code below shows this for the black and white postscript device.

Chapter 3 Traditional Trellis Graphics

```
> trellis.device(postsript)
> Rows(trellis.par.get("superpose.symbol"),1:2)

$cex:
[1] 0.85 0.85

$col:
[1] 1 1

$font:
[1] 1 1

$pch:
[1] "\001" "+"

> dev.off()
null device
      1
```

Note that only two values are returned for each graphical parameter, instead of the usual seven.

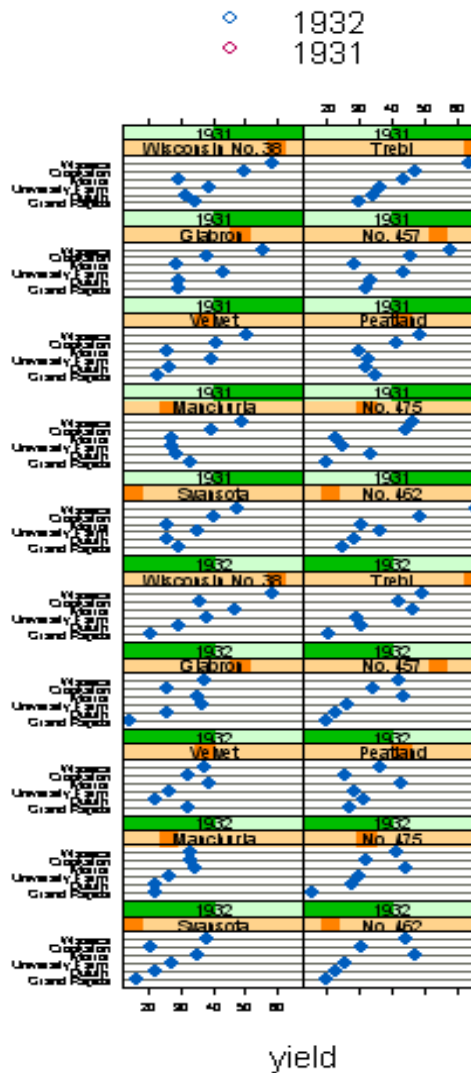


Figure 3.30: Multipanel dot plot of the barley data. A key is included to distinguish the plotting symbols.

The key for the `barley.plot` object has two entries, one for each year. Each entry has two items, the order of which is determined by the order in the key argument. In the call to `update` above, `points` is specified first and `text` is second in the key argument; thus, plotting symbols are displayed first in the key, and `text` is second.

By default, the two entries in our key are drawn as an array with one column and two rows. We can change this with the `columns` component of the `key` argument. The following command illustrates this, and also switches the order of the symbols and the text:

```
> update(barley.plot, key = list(
+   text=list(levels(barley$year)),
+   points=Rows(trellis.par.get("superpose.symbol"),1:2),
+   columns=2))
```

Repositioning a Key

If the default location of a key seems a bit too far from the rest of the graph, the key can be repositioned with a combination of various components in the `key` argument. The code below shows one approach for accomplishing this.

```
> update(barley.plot, key = list(
+   points=Rows(trellis.par.get("superpose.symbol"),1:2),
+   text=list(levels(barley$year)),
+   columns=2,
+   border=1,
+   space="top",
+   x=0.5,
+   y=1.02,
+   corner=c(0.5,0)))
```

For clarity, this command uses the `border` argument to draw a border around the key. The `border` argument accepts a number that specifies the color in which the border should be drawn.

To reposition a key, two coordinate systems are required. The first describes locations in the rectangle that encloses the panels of the display, but does not include the tick marks. The lower left corner of this rectangle has coordinates 0,0 and the upper right corner has coordinates 1,1. A location in the panel rectangle is specified by the components `x` and `y` in the `key` argument. In the command above, `x=0.5` and `y=1.02`, which centers the key horizontally and places it at the top of the figure, just outside the rectangle.

The second coordinate system describes locations in the border rectangle of the key, which is shown when the border is drawn. The lower left corner of the border rectangle has coordinates 0,0 and the upper right corner has coordinates 1,1. A location in the border rectangle is specified by the `corner` component, which is a vector with two elements specifying the horizontal and vertical coordinates. The

key is positioned so that the locations specified by the two coordinate systems are at the same place on the graph. Having two coordinate systems makes it far easier to place a key quickly, often on the first try.

The space component of the key argument allocates space for the key in the margins. It takes one of four values and allocates space on the corresponding side of the graph: "top", "bottom", "right", and "left". By default, space for a key is allocated at the top of a graph. However, notice that we explicitly specified `space="top"` in the command above. The reason is that as soon as the components `x`, `y`, and `corner` are specified, no default space is allocated in any margin location unless we explicitly use `space`.

To allocate space on the right side of the graph, type:

```
> update(barley.plot, key = list(
+   points=Rows(trellis.par.get("superpose.symbol"),1:2),
+   text=list(levels(barley$year)),
+   space="right"))
```

To position the key in the upper left corner of the border rectangle, at the same vertical position as the top of the panel rectangle and at a horizontal position slightly to the right of the right side of the panel rectangle, type:

```
> update(barley.plot, key = list(
+   points=Rows(trellis.par.get("superpose.symbol"),1:2),
+   text=list(levels(barley$year)),
+   space="right",
+   border=1,
+   corner=c(0,1),
+   x=1.05,
+   y=1))
```

For clarity, this command also draws a border around the key.

Including Lines in a Key

So far this section, we have seen that the components `points` and `text` can be used to create items in key entries. A third component, `lines`, draws line items. To illustrate this, let us return to the `fuel.frame` data. The following code creates a plot of the `Mileage` variable against `Weight` for the six types of cars, and adds two loess smooths using different values of the smoothing parameter `span`:

```
> superpose.line <- trellis.par.get("superpose.line")
```

```
> superpose.symbol <- trellis.par.get("superpose.symbol")
> xyplot(Mileage~Weight, data=fuel.frame, groups=Type,
+ aspect=1, panel=function(x,y,...) {
+   panel.superpose(x,y,...)
+   panel.loess(x,y, span=1/2,
+     lwd=superpose.line$lwd[1],
+     lty=superpose.line$lty[1],
+     col=superpose.line$col[1])
+   panel.loess(x,y, span=1,
+     lwd=superpose.line$lwd[2],
+     lty=superpose.line$lty[2],
+     col=superpose.line$col[2]) },
+ key = list(transparent=T, x=0.95, y=0.95, corner=c(1,1),
+   lines=Rows(superpose.line,1:6),
+   size=c(3,3,0,0,0,0),
+   text=list(c("Span = 0.5", "Span = 1.0", rep("",4))),
+   points=Rows(superpose.symbol,1:6),
+   text=list(levels(fuel.frame$Type))))
```

ASPECT RATIO

2D Displays

The *aspect ratio* of a two-dimensional graph is the height of a data region divided by its width. Aspect ratio is a critical factor in determining how well a display shows the structure of the data. There are situations where varying the aspect ratio reveals information in the data that cannot be seen if the graph is square. This often occurs when the aspect ratio is chosen to bank the underlying pattern of points to 45 degrees.

More generally, any time we graph a curve or a scatter of points, controlling the aspect ratio is vital. One important feature of Trellis Graphics is the direct control of the aspect ratio through the argument `aspect`. You can use the `aspect` argument with most general display functions to set the ratio to a specific value. In Figure 3.31, for example, the aspect ratio has been set to 3/4:

```
> xyplot(N0x~E, data=gas, aspect=3/4)
```

If `aspect="xy"`, line segments are banked to 45 degrees. Here is how it works: suppose x and y are data points to be plotted, and consider the line segments that connect successive points. The aspect ratio is chosen so that the absolute values of the slopes of these segments are centered at 45 degrees. This is done in Figure 3.32 with the expression

```
> xyplot(N0x~E, data=gas, aspect="xy")
```

The resulting aspect ratio is approximately 0.4. Ordinarily, you should bank line segments based on a smooth underlying pattern in the data. That is, you should bank based on the line segments of a fitted curve, instead of arbitrarily setting the `aspect` argument. You can do this with Trellis Graphics, as shown in the section Prepanel Functions (page 191).

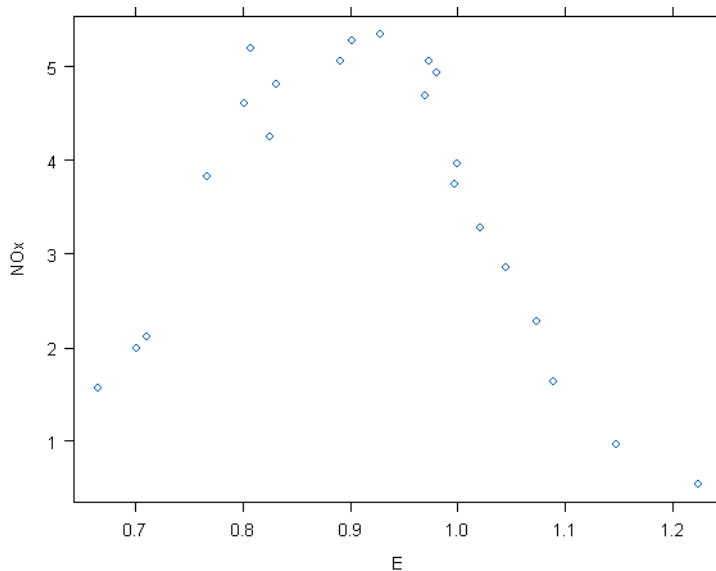


Figure 3.31: Scatter plot of the gas data with an aspect ratio of 3/4.

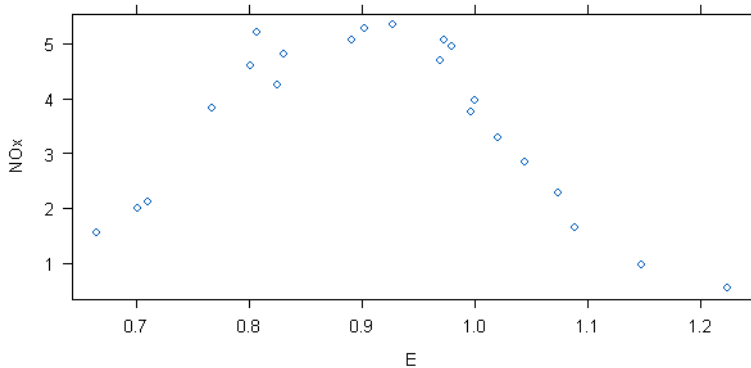


Figure 3.32: Scatter plot of the gas data with line segments banked to 45 degrees.

3D Displays

The aspect ratio of a three-dimensional plot is defined by two ratios: the length of the y axis to the length of the x axis, and the length of the z axis to the length of the x axis. Thus, the aspect argument to the 3D general display functions `wireframe` and `cloud` accepts a numeric

vector of length two. For example, the following command displays a surface plot of the `gauss` data introduced in the section Visualizing Three-Dimensional Data (page 136):

```
> wireframe(dataz ~ datax*datay, data=gauss, aspect=c(1,2))
```

In the resulting figure, the y axis and the x axis are equal in length, and the z axis is twice as long as both of them. For more details on the `aspect` argument in three-dimensional displays, see the help file for `trellis.3d.args`.

Prepanel Functions

Banking to 45 degrees is an important display method built into Trellis Graphics through the argument `aspect`. In addition, axis ranges in a Trellis graph can be controlled by the arguments `xlim`, `ylim`, and `scales` introduced in the section Scales and Labels (page 166). Another argument, `prepanel`, is a function that supplies information for the banking and range calculations.

For example, the code below plots the `ethanol` data. In the plot, the variable `NOx` is graphed against `E` given `C`, and loess curves are superposed in each panel.

```
> xyplot(NOx ~ E|C, data=ethanol, aspect=1/2,
+   panel = function(x,y) {
+     panel.xyplot(x,y)
+     panel.loess(x, y, span=1/2, degree=2)
+   }
+ )
```

The result is shown in Figure 3.33. We would like to do two things with this plot: one involves the aspect ratio and the other involves axis ranges. We use the `prepanel` argument to `xyplot` to accomplish both of these things.

First, consider the argument value `aspect=1/2` in the above command. We could have set `aspect="xy"` to bank the line segments connecting the graphed values of `E` and `NOx` to 45 degrees. Normally, however, we do not want to carry out banking of the raw data if they are noisy. Rather, we want to bank an underlying smooth pattern in this situation. In the examples below, we show how to bank using the line segments of the loess curves.

Second, notice that the loess curve exceeds the maximum value of the vertical axis in the top panel. This occurs because Spotfire S+ chooses the axis limits based on the values of E and NOx, but the loess curves are computed by `panel.loess` after all of the scaling has been carried out. In the examples below, we show how to force the scaling to account for the values of the loess curves.

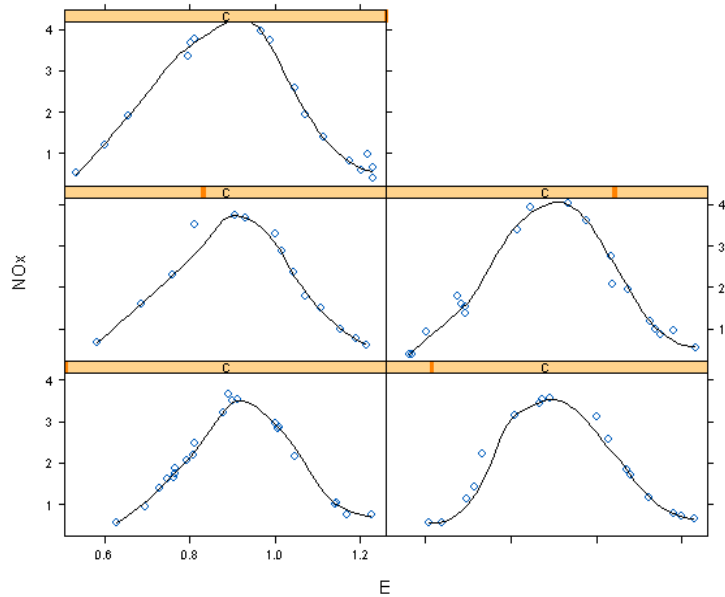


Figure 3.33: Multipanel display of the ethanol data with loess curves superposed.

The argument `prepanel` to the general display functions allows us to bank data points to 45 degrees based on loess curves. In addition, we can take the loess curves into account when computing the axis ranges. The following command shows how to do this:

```
> xyplot(NOx ~ E|C, data = ethanol,
+   prepanel = function(x,y)
+     prepanel.loess(x, y, span=1/2, degree=2),
+   layout = c(1,6),
+   panel = function(x,y) {
+     panel.xyplot(x,y)
+     panel.loess(x, y, span=1/2, degree=2)
+   }
+ )
```

The `prepanel` argument accepts a function and does panel-by-panel computations, just like the `panel` argument. However, the `prepanel` computations are carried out before the axis limits and aspect ratio are determined; this allows them to be used in the determination of the axis scaling.

The return value of a `prepanel` function is a list with prescribed component names. These names are shown in the code of `prepanel.loess`:

```
> prepanel.loess
function(x, y, ...)
{
  xlim <- range(x)
  ylim <- range(y)
  out <- loess.smooth(x, y, ...)
  x <- out$x
  y <- out$y
  list(xlim = range(x, xlim), ylim = range(y, ylim),
       dx = diff(x), dy = diff(y))
}
```

The component values `xlim` and `ylim` determine axis ranges just as they do when given as arguments to the general display functions. The values `dx` and `dy` are the horizontal and vertical changes of the line segments that are to be banked to 45 degrees.

The function `prepanel.loess` computes the smooths for all panels. It then computes values of `xlim` and `ylim` to ensure the curves are included in the ranges of the axes. Finally, `prepanel.loess` returns the changes in the line segments that make up the plotted curve. Any of the component names can be missing from the list; if either `dx` or `dy` is missing, the other must be as well. When `dx` and `dy` are both present, they provide the information needed for banking to 45 degrees, as well as the instruction to do so. Thus, the `aspect` argument should not be used as an argument when `dx` and `dy` are present.

DATA STRUCTURES

Trellis Graphics uses the Spotfire S+ formula language to specify data for plotting. This requires the data to be stored in structures that work with formulas. Roughly speaking, the data variables must either be stored in a data frame or be vectors of the same length; this is also true of the Spotfire S+ modeling functions such as `lm`.

To ensure that Trellis functions are easy to use regardless of the structure of your data, Spotfire S+ includes three functions that convert data structures of different types into data frames. The `make.groups` converts multiple vectors into a single data frame, the `as.data.frame.array` function converts multidimensional arrays into data frames, and `as.data.frame.ts` converts time series into data frames. We discuss each of these functions in this section.

Vectors

The function `make.groups` takes several vectors and constructs a data frame with two variables, `data` and `which`. To illustrate this function, consider payoffs of the New Jersey Pick-It lottery from three time periods. The data are stored in the built-in vectors `lottery.payoff`, `lottery2.payoff`, and `lottery3.payoff`. Suppose we want to create box plots of the vectors to compare their distributions. We first convert the three vectors to a single data frame using the `make.groups` function:

```
> lottery.payoffs <- make.groups(  
+   "1975" = lottery.payoff,  
+   "1977" = lottery2.payoff,  
+   "1981" = lottery3.payoff)
```

```
> lottery.payoffs
```

```
      data which  
1 190.0  1975  
2 120.5  1975  
3 285.5  1975  
4 184.0  1975  
5 384.5  1975  
6 324.5  1975  
7 114.0  1975  
8 506.5  1975
```

```

 9 290.0 1975
10 869.5 1975
11 668.5 1975
12  83.0 1975
13 . . .

```

The `data` column is a numeric variable containing the payoff values from each of the three vectors. The `which` column is a factor variable with three levels corresponding to the chosen names "1975", "1977", and "1981":

```

> names(lottery.payoffs)
[1] "data" "which"

> levels(lottery.payoffs$which)
[1] "1975" "1977" "1981"

```

Thus, `lottery.payoff` appears at the beginning of the data frame, `lottery2.payoff` is in the middle, and `lottery3.payoff` is at the end. Note that it is not necessary to specify names for the factor levels in `make.groups`; if names are not given, the levels correspond to the names of the original data vectors.

The command below shows how to create box plots of the `lottery.payoffs` data:

```

> bwplot(which~data, data=lottery.payoffs)

```

Arrays

The function `as.data.frame.array` converts multidimensional arrays into data frames. Consider the built-in object `iris`, which is a three-way array containing 50 measurements of four variables for each of three varieties of iris:

```

> dim(iris)
[1] 50 4 3

```

To convert `iris` to a data frame in preparation for Trellis plotting, use the `as.data.frame.array` function as follows:

```

> iris.df <- as.data.frame.array(iris, col.dims=2)
> names(iris.df) <- c("Sepal.L", "Sepal.W",
+   "Petal.L", "Petal.W", "flower", "variety")

```

The resulting data frame `iris.df` has the second dimension of `iris` as its first four columns:

```
> iris.df
      Sepal.L Sepal.W Petal.L Petal.W flower  variety
1         5.1    3.5    1.4    0.2     1    Setosa
2         4.9    3.0    1.4    0.2     2    Setosa
3         4.7    3.2    1.3    0.2     3    Setosa
4         4.6    3.1    1.5    0.2     4    Setosa
5         5.0    3.6    1.4    0.2     5    Setosa
6 . . . .
```

To produce a scatterplot matrix of the `iris.df` data, use the following code:

```
> superpose.symbol <- trellis.par.get("superpose.symbol")
> for (i in 1:4) iris.df[,i] <- jitter(iris.df[,i])

> splom(~iris.df[,1:4],
+       key = list(
+         space="top", columns=3,
+         text=list(levels(iris.df$variety)),
+         points=Rows(superpose.symbol, 1:3)),
+       groups = iris.df$variety,
+       panel = panel.superpose)
```

To prevent the plotting symbols from overlapping, the data have been jittered using the `jitter` function.

Time Series

The function `as.data.frame.ts` accepts one or more time series as arguments and produces a data frame with variables named `series`, `which`, `time`, and `cycle`. The `series` component is the data from all of the time series combined into one long vector. The `time` component gives the time associated with each of the data points, measured in the same time units as the original series. The `cycle` variable gives the periodic component of the times, and `which` is a factor that identifies the original time series containing the measurement.

In the following example, we use `as.data.frame.ts` to convert the built-in time series `hstart`. The `hstart` series contains census data on the number of housing starts in the United States from January 1966 to December 1974.

```
> as.data.frame.ts(hstart)

  series  which    time cycle
1  81.9  hstart 1966.000   Jan
2  79.0  hstart 1966.083   Feb
3 122.4  hstart 1966.167   Mar
4 143.0  hstart 1966.250   Apr
5 133.9  hstart 1966.333   May
6 . . .
```

The following command displays the housing starts for each month separately:

```
> xyplot(series ~ time|cycle,
+   data = as.data.frame.ts(hstart), type = "b",
+   xlab = "Year", ylab = "Housing Starts by Month")
```

SUMMARY OF TRELIS FUNCTIONS AND ARGUMENTS

Table 3.1: *An alphabetical guide to Trellis Graphics.*

Statement	Purpose	Example
<code>as.data.frame.array</code>	function	<code>iris.df <- as.data.frame.array(iris, col.dims = 2)</code>
<code>as.data.frame.ts</code>	function	<code>data.frame.ts(hstart)</code>
<code>aspect</code>	argument	<code>xyplot(NOx~E, data=gas, aspect=1/2, xlab = "Equivalence Ratio", ylab = "Oxides of Nitrogen", main = "Air Pollution", sub = "Single-Cylinder Engine")</code>
<code>barchart</code>	function	<code>barchart(names(mileage.means) ~ mileage.means, aspect = 1)</code>
<code>between</code>	argument	<code>barley.plot <- update(barley.plot, between = list(y=c(0,0,0,0,1,0,0,0)))</code>
<code>bwplot</code>	function	<code>bwplot(Type~Mileage, data=fuel.frame, aspect=1)</code>
<code>cloud</code>	function	<code>cloud(Mileage ~ Weight*Disp., data = fuel.frame, screen = list(z=-30, x=-60, y=0), xlab = "W", ylab = "D", zlab = "M")</code>
<code>contourplot</code>	function	<code>contourplot(dataz ~ datax*datay, data = gauss, aspect = 1, at = seq(from=0.1, to=0.9. by=0.2))</code>
<code>data</code>	argument	See the aspect example.
<code>densityplot</code>	function	<code>densityplot(~Mileage, data=fuel.frame, aspect=1/2, width=5)</code>
<code>dev.off</code>	function	<code>dev.off()</code>

Table 3.1: An alphabetical guide to Trellis Graphics. (Continued)

Statement	Purpose	Example
<code>dotplot</code>	function	<code>dotplot(names(mileage.means) ~ logb(mileage.means, base=2), aspect=1, cex=1.25)</code>
<code>equal.count</code>	function	<code>GIVEN.E <- equal.count(ethanol\$E, number=9, overlap=1/4)</code>
<code>formula</code>	argument	<code>xyplot(formula = gas\$NOx ~ gas\$E)</code>
<code>histogram</code>	function	<code>histogram(~Mileage, data=fuel.frame, aspect=1, nint=10)</code>
<code>intervals</code>	argument	<code>GIVEN.E2 <- shingle(ethanol\$E, intervals = cbind(endpoints[-6], endpoints[-1]))</code>
<code>jitter</code>	argument	<code>stripplot(Type~Mileage, data=fuel.frame, jitter=TRUE, aspect=1)</code>
<code>key</code>	argument	<code>update(barley.plot, key = list(points = Rows(trellis.par.get("superpose.symbol"), 1:2), text = list(levels(barley\$year))))</code>
<code>layout</code>	argument	<code>dotplot(site ~ yield year*variety, data=barley, layout=c(2,5,2))</code>
<code>levelplot</code>	function	<code>levelplot(dataz ~ datax*datay, data=gauss, aspect=1, cuts=6)</code>
<code>levels</code>	function	<code>levels(barley\$year)</code>
<code>main</code>	argument	See the aspect example.
<code>make.groups</code>	function	<code>lottery.payoffs <- make.groups(lottery.payoff, lottery2.payoff, lottery3.payoff)</code>

Table 3.1: An alphabetical guide to Trellis Graphics. (Continued)

Statement	Purpose	Example
<code>page</code>	argument	<code>update(market.plot, page = function(n) text(x=0.75, y=0.95, paste("page",n), adj=0.5))</code>
<code>panel</code>	argument	<code>xyplot(NOx~E, data=gas, aspect=1/2, panel=panel.special)</code>
<code>panel.superpose</code>	function	<code>xyplot(Mileage~Weight, data=fuel.frame, aspect=1, groups=Type, panel=panel.superpose)</code>
<code>panel.loess</code>	function	<code>xyplot(NOx ~ C GIVEN.E, data=ethanol, aspect=2.5, panel=function(x,y) { panel.xyplot(x,y) panel.loess(x, y, span=1) })</code>
<code>panel.xyplot</code>	function	See the <code>panel.loess</code> example
<code>parallel</code>	function	<code>parallel(~fuel.frame)</code>
<code>par</code>	function	<code>par(ask=TRUE)</code>
<code>par.strip.text</code>	argument	<code>xyplot(NOx ~ E C, data = ethanol, par.strip.text = list(cex=2))</code>
<code>piechart</code>	function	<code>piechart(names(mileage.means) ~ mileage.means)</code>
<code>prepanel</code>	argument	<code>xyplot(NOx ~ E C, data=ethanol, prepanel = function(x,y) prepanel.loess(x, y, span=1/2, degree=2), layout=c(1,6), panel=function(x,y) { panel.xyplot(x,y) panel.loess(x, y, span=1/2, degree=2) })</code>
<code>prepanel.loess</code>	function	See the <code>prepanel</code> example
<code>print</code>	function	<code>print(box.plot, position=c(0, 0, 1, 0.4), more=TRUE)</code>

Table 3.1: An alphabetical guide to Trellis Graphics. (Continued)

Statement	Purpose	Example
<code>print.trellis</code>	function	<code>?print.trellis</code>
<code>qq</code>	function	<code>qq(Type~Mileage, data=fuel.frame, aspect=1, subset = (Type=="Compact") (Type=="Small"))</code>
<code>qqmath</code>	function	<code>qqmath(~Mileage, data = fuel.frame, subset = (Type=="Small"))</code>
<code>reorder.factor</code>	function	<code>barley\$variety <- reorder.factor(barley\$variety, barley\$yield, median)</code>
Rows	function	<code>Rows(trellis.par.get("superpose.symbol"), 1:2)</code>
<code>scales</code>	argument	<code>xyplot(NOx~E, data = gas, aspect = 1/2, ylim = c(0,6), scales = list(cex=2, tick number=4))</code>
<code>screen</code>	argument	<code>wireframe(dataz ~ datax*datay, data = gauss, drape = FALSE, screen = list(z=45, x=-60, y=0))</code>
<code>shingle</code>	function	See the intervals example.
<code>show.settings</code>	function	<code>show.settings()</code>
<code>skip</code>	argument	<code>bwplot(age ~ log(1+usage) income*pick, strip = function(...) strip.default(..., strip.names=TRUE), skip = c(F,F,F,F,F,F,T), layout = c(2,4,2), data = market.survey)</code>
<code>span</code>	argument	See the <code>prepanel.loess</code> example.

Table 3.1: An alphabetical guide to Trellis Graphics. (Continued)

Statement	Purpose	Example
<code>space</code>	argument	<code>update(barley.plot, key = list(points = Rows(trellis.par.get("superpose.symbol"), 1:2), text = list(levels(barley\$year)), space = "right"))</code>
<code>splom</code>	function	<code>splom(~fuel.frame)</code>
<code>strip</code>	argument	See the skip example.
<code>stripplot</code>	function	See the jitter example.
<code>sub</code>	argument	See the aspect example.
<code>subscripts</code>	argument	<code>xyplot(N0x ~ E C, data = ethanol, aspect = 1/2, panel = function(x,y,subscripts) text(x, y, subscripts, cex=0.75))</code>
<code>subset</code>	argument	<code>xyplot(N0x~E, data = gas, subset = E<1.1)</code>
<code>superpose.symbol</code>	argument	<code>trellis.par.get("superpose.symbol")</code>
<code>trellis.args</code>	arguments	<code>?trellis.args</code>
<code>trellis.3d.args</code>	arguments	<code>?trellis.3d.args</code>
<code>trellis.device</code>	function	<code>trellis.device(postscript, file = "settings.ps")</code>
<code>trellis.par.get</code>	function	<code>plot.line <- trellis.par.get("plot.line")</code>
<code>trellis.par.set</code>	function	<code>trellis.par.set("plot.symbol", plot.symbol)</code>
<code>update</code>	function	See the between example.

Table 3.1: An alphabetical guide to Trellis Graphics. (Continued)

Statement	Purpose	Example
<code>width</code>	argument	See the <code>densityplot</code> example.
<code>wireframe</code>	function	See the <code>screen</code> example.
<code>xlab</code>	argument	See the <code>aspect</code> example.
<code>xlim</code>	argument	<code>xyplot(NOx~E, data = gas, xlim = c(0,2))</code>
<code>xyplot</code>	function	<code>xyplot(Mileage~Weight, data=fuel.frame, aspect=1)</code>
<code>ylab</code>	argument	See the <code>aspect</code> example.
<code>ylim</code>	argument	See the <code>scales</code> example.

EDITING GRAPHICS IN UNIX

4

Introduction	206
Basic Terminology	206
Using motif Graphics Windows	207
Starting and Stopping the motif Device	207
An Example Plot	207
Motif Window Features	209
The Options Menu	211
Available Colors Under X11	216
Using java.graph Windows	221
Starting and Stopping the java.graph Device	221
An Example Plot	221
java.graph Window Features	223
The Options Menu	224
Printing Your Graphics	233
Printing with PostScript Printers	233
Using the Print Option from the Motif Window	234
Print Options in the Java Graphics Window	235
Using the printgraph Function	235
Using the postscript Function	236
Creating Encapsulated PostScript Files	239
Setting PostScript Options	241
Creating Color PostScript Graphics	243
Creating Bitmap Graphics	245
Managing Files from Hard Copy Graphics Devices	246
Using Graphics from a Function or Script	246

INTRODUCTION

In this section, we assume you are familiar with your particular window system. In particular, we assume you know how to start your window system and set your display so that X11 applications can display windows on your screen. For further information on a particular window system, consult your system administrator or the following references:

- Quercia, V. and O'Reilly, T. (1989). *X Window System User's Guide*. Sebastopol, California: O'Reilly and Associates.
- Quercia, V. and O'Reilly, T. (1990). *X Window System User's Guide, Motif Edition*. Sebastopol, California: O'Reilly and Associates.

Basic Terminology

In this section, we refer to the window in which you start Spotfire S+ as the *Spotfire S+ window*. The window that is created when you start a windowing graphics device from the Spotfire S+ window is called the *graphics window*.

USING MOTIF GRAPHICS WINDOWS

In a `motif` graphics window, you can interactively change the color specifications of your plots and immediately see the result and also interactively change the specifications that are used to send the plot to a printer.

Starting and Stopping the motif Device

To open a `motif` graphics window, start the `motif` device by calling the `motif` function from Spotfire S+ command prompt:

```
> motif()
```

For information about the available `motif` arguments, see the section The `motif` Device on page 364 and refer to `motif` in the Spotfire S+ online help.

The `motif` device is started automatically in both the Java-enabled and Java-disabled command-line versions of Spotfire S+ if no other graphics device is open when you use Spotfire S+ to evaluate a high-level plotting function.

To stop a `motif` graphics device (and close the `motif` window) without quitting Spotfire S+, use the `dev.off` or `graphics.off` function.

Warning

Do not remove the `motif` graphics window by using a window manager menu! If you remove a `motif` window in this way, Spotfire S+ will not know that the graphics device has been removed. Thus, this graphics device will still appear on the vector returned by `dev.list`, but if you try to send plot commands to it you will get an error message. If you do accidentally remove the `motif` window with a window manager menu, use the `dev.off` function to tell Spotfire S+ that this device is no longer active.

An Example Plot

As you explore the various `motif` features, you can use the following S-PLUS code to generate an easily-reproducible graphic:

```
> plot(corn.rain, corn.yield, type="n",
+      main="Plot Example")
> points(corn.rain, corn.yield, pch="*", col=2)
> lines(lowess(corn.rain, corn.yield), lty=2, col=3)
```

```
> legend(12, 23, c("Color 1", "Color 2", "Color 3"),  
+       pch=" * ", lty=c(1, 0, 2), col=c(1, 2, 3))
```

Note that in the call to `legend` there is a space before and after the `*` in the argument `pch=" * "`.

To create the example plot in a `motif` window, first start the `motif` device:

```
> motif()
```

Then run the example code to create the plot shown in Figure 4.1.

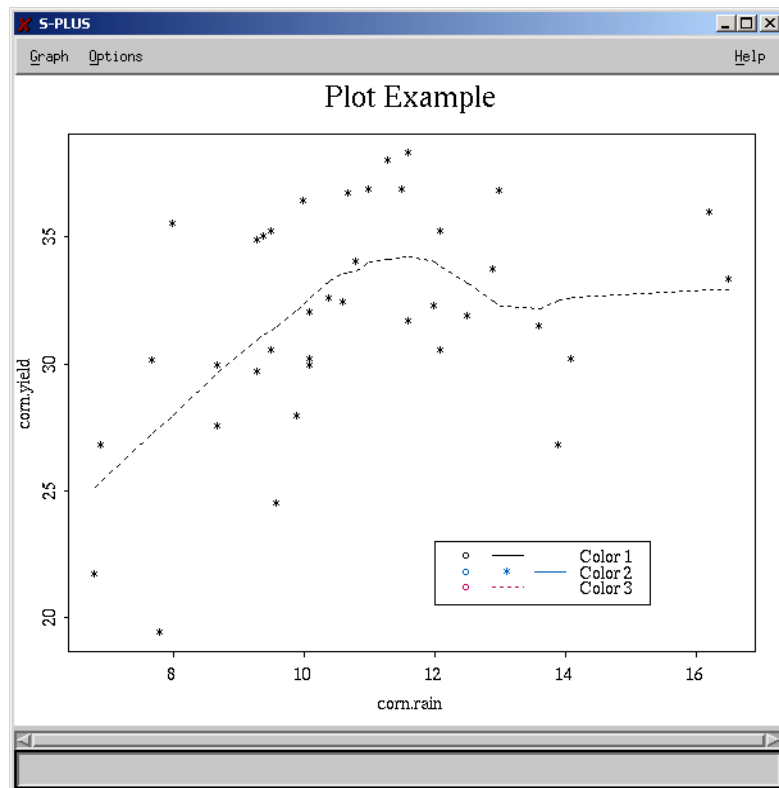


Figure 4.1: *Plot example in a motif window.*

By default, the color of the title, legend box, axis lines, axis labels, and axis titles are color 1. We have specified the points to have color 2, and the dashed line representing the smooth from the `lowess` command to have color 3.

Motif Window Features

The features of the Spotfire S+ `motif` window are described below.

- *Title bar* Contains the window **Menu** button, the title Spotfire S+, the **Minimize** button, and the **Maximize** button.
- *Menu Bar* Contains three menu titles: **Graph**, **Options**, and **Help**.
- *Pane* Area where Spotfire S+ displays any graphs that you create while the `motif` graphics device is active.
- *Footer* Area where Spotfire S+ displays status or error messages about the graph you have created.
- *Resize Borders* Used to change the size of the window.

The Help Menu

The **Help** menu at the far right side of the menu bar produces a pop-up window, rather than a menu, when you select it. The help window contains a condensed version of the `motif` help file.

The Graph Menu

The **Graph** menu provides the following commands:

- *Redraw* Redraws the graph that appears in the pane of the graphics window.
- *Copy* Creates a copy of the current graphics window, as shown in Figure 4.2. The copy has a title bar, a menu bar, a pane, and a footer, just like the original. The title in the title area is **Spotfire S+ Copy**. The menu bar in a copy of the graphics window does not contain an Options menu title, only the Graph and Help menu titles.
- *Print* Converts the current plot in the graphics window to either a PostScript or LaserJet file and then sends the file to your printer. Selecting Print is not equivalent to typing the `printgraph()` command in the Spotfire S+ window. The `printgraph` command uses Spotfire S+ environment variables to

determine printing defaults, whereas Print uses the specifications shown in the **Printing** dialog box.

When you select **Print**, a message is displayed in the footer of the graphics window telling you what kind of file was created and the command that was used to route the file to the printer. See the section The Options Menu (page 211) for a description of how to set the defaults for printing.

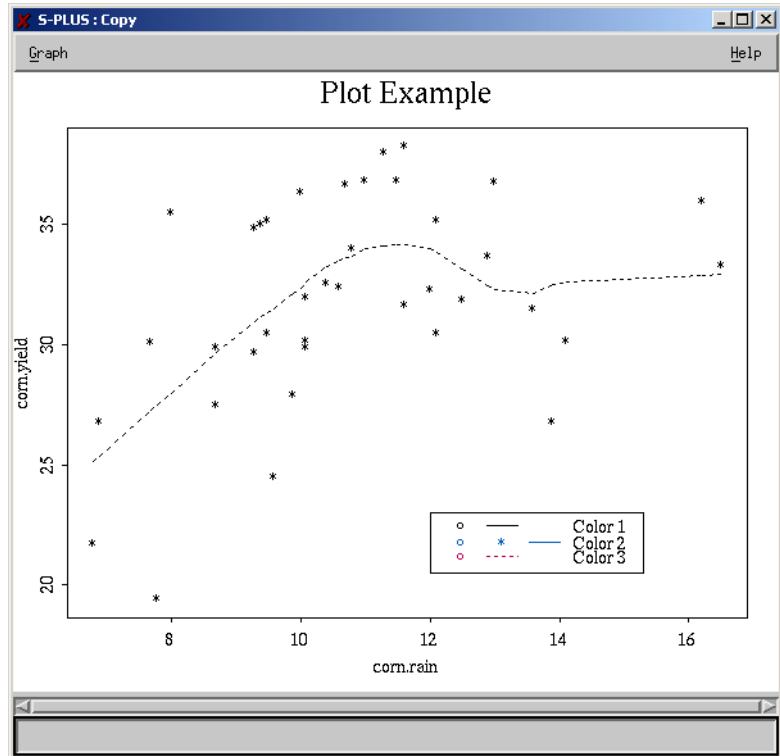


Figure 4.2: A copy of the *motif* graphics window.

The Options Menu

The Options menu provides two commands: Color Scheme and Printing.

Note

The information in the following “Color Scheme Dialog Box” section applies *only* if you are using Spotfire S+ in the backward compatible `use.device.palette(T)` mode. When using Spotfire S+ in `use.device.palette(F)` mode (which is the default), disregard this information and refer to section Color Specification on page 3 for information about working with colors.

For information about backward compatibility for the Spotfire S+ graphics features, see section Backward Compatibility (page 24).

Color Scheme Dialog Box

The Color Scheme dialog box is a powerful feature of the motif windowing graphics device. It allows you to change the colors in your plot interactively and immediately see the results. Figure 4.3 shows an example of the Color Scheme dialog box. This window has a menu button in title bar.

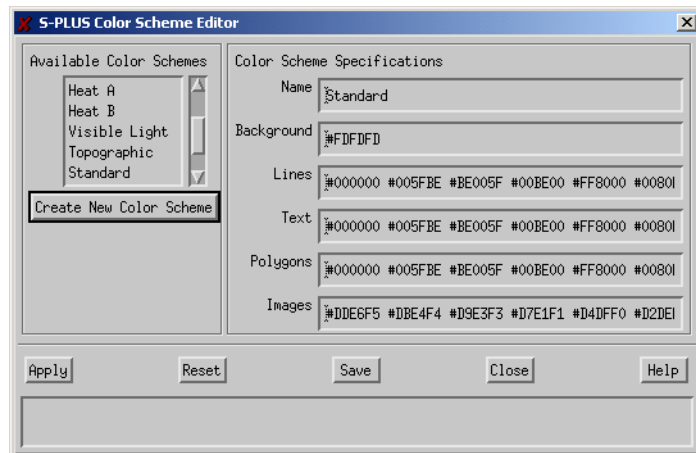


Figure 4.3: The *motif* Color Scheme dialog box.

The Color Scheme dialog contains:

- The Available Color Schemes list
- The Color Scheme Specifications editor showing the specifications for the selected color scheme

- A button marked Create New Color Scheme
- An Apply button
- A Reset button
- A Save button
- A Close button
- A Help button

The Help Button

The **Help** button displays a pop-up help window for this dialog box.

The Color Scheme Specifications Editor

The **Color Scheme Specifications** editor includes specifications for the following characteristics:

- *Name* The name of the color scheme.
- *Background* The color of the background. This specification can have only one color name or value.
- *Lines* The color names or values used for lines.
- *Text* The color names or values used for text.
- *Polygons* The color names or values used with the polygon, pie, barplot, and hist plotting functions.
- *Images* The color names or values used with the image plotting function.

All color schemes must have values for the specifications **Name**, **Background**, and **Lines**. The specifications for **Text**, **Polygons**, and **Images** default to the specifications for **Lines** if left blank.

See the section Available Colors Under X11 (page 216) for information on how to specify colors with the `motif` windowing graphics device.

Applying a Different Color Scheme

To apply a different color scheme, select one of the color scheme names under the **Available Color Schemes** option menu. Note that its specifications are displayed in the **Color Scheme Specifications** editor. The plot in the graphics window, however, is still based on the original color scheme. To apply the selected color scheme, click the **Apply** button.

Your available color schemes will not necessarily have the names or specifications shown in Figure 4.3. Initially, the available color schemes are defined using X resources.

Creating New Color Schemes

To create a new color scheme, follow these steps:

1. Click the **Create New Color Scheme** button. The word “unnamed” appears as the last color scheme in the **Available Color Schemes** list. The default values under the **Color Scheme Specifications** are the name “unnamed”, a black background, and white lines.
2. Click the **Name** box. Type in a name to replace “unnamed”.
3. In the **Background** box, specify the background color for this color scheme. The background can only have one color value. Refer to the section Available Colors Under X11 (page 216) for information on available color names.
4. In the **Lines** box type in one or more color names.
5. Repeat the previous step for the **Text, Polygons, and Images** boxes.
6. To make this color scheme permanent, click the **Save** button. If you do not save your newly-created color scheme, it remains only until you close the graphics window.
7. Click the **Apply** button. The plot in the graphics window is now based on your newly-created color scheme.

The Reset Button

Any time you are using the **Color Scheme** dialog box, click the **Reset** button. If you have not yet clicked on the **Apply** button, then the **Available Color Schemes** menu and **Color Scheme Specifications** editor are set to how they were when you first entered

the dialog box. If you have at some time clicked on the **Apply** button, then the color schemes are reset to how they were immediately after the last time you clicked on the **Apply** button.

The Printing Dialog Box

The **Printing** command in the **Options** menu allows you to interactively change the specifications for the printing method used when you select **Print** from the **Graph** menu.

Figure 4.4 shows the **Printing** dialog box, which provides options for the printing **Method** and **Orientation**, a text entry box for specifying the actual print **Command**, and (if **Method** is **LaserJet**) options for specifying the **Resolution**. There are buttons labeled **Apply**, **Reset**, **Print**, **Save**, **Close**, and **Help**. This window also has a menu button in the title bar.

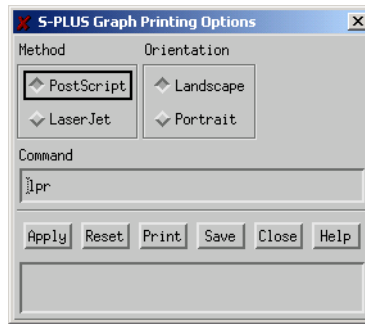


Figure 4.4: The *motif* Printing dialog box.

Method, Orientation, Resolution, and Command

The print options are as follows:

- *Method* Determines the kind of file that is created when you select the **Print** command. The **PostScript** method produces a file compatible with PostScript printers. The **LaserJet** method produces a file of compatible with LaserJet printers.
- *Orientation* Determines the orientation of the graph on the paper. **Landscape** orientation puts the x -axis along the

long side of the paper; **Portrait** orientation puts the x -axis along the short side of the paper.

- *Command* Allows you to specify the command that is used to send the file to the printer.
- *Resolution* Available only if **Method** is set to **LaserJet**. Allows you to specify the resolution of plots printed on an HP LaserJet printer.

The default settings for **Method**, **Orientation**, **Command**, and **Resolution** are initially set using X resources. The way to change these settings is explained below.

Printing Options Buttons

- *Apply* Click to apply any changes you have made to the printing specifications. Only the specifications are changed; no printing is done. Any changes you make last only as long as the graphics window remains, or until you make more changes and select **Apply** again. Once you close the graphics window, any changes to the original default settings are lost unless you click the **Save** button.
- *Reset* Click to reset the printing specifications. If you have not yet clicked the **Apply** button, then the specifications are set to how they were when you first entered the dialog box. If you have at some time clicked the **Apply** button, then the specifications are reset to their state after the last time you clicked the **Apply** button.
- *Print* Click to apply any printing specification changes you have made *and* send the graph to the printer.

- *Save* Click to save the current printing specifications as the default.
- *Close* Click to dismiss the dialog box.
- *Help* Click to pop-up a **Help** window for this dialog box.

Figure 4.5 shows how the **Printing** dialog box in Figure 4.4 changes when you select the **LaserJet** method. The **Resolution** options appear, and the **Command** line changes.

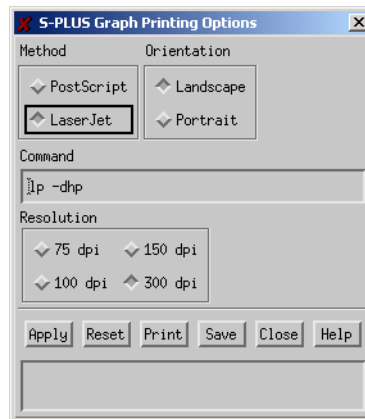


Figure 4.5: Changing printing methods.

Available Colors Under X11

To specify color schemes for the `motif` device, use the **Color Scheme Specifications** window.

To specify a color scheme, you must create a list of colors. There are two ways to list colors in a color scheme:

- Use color *names* listed in the system file `rgb.txt`.
- Use hexadecimal *values* that represent colors in the RGB Color Model.

The first method is a “front end” to the second method; it is easier to use, but you are limited to the colors listed in the `rgb.txt` file. The second method is more complex, but it allows you to specify any color your display is capable of producing.

The initial set of colors is set system-wide at installation. Any changes you make using the **Color Scheme Specifications** window override the system values. This remains true even if system-wide changes are installed.

Viewing Color Names in `rgb.txt`

The `rgb.txt` file contains a list of predefined colors that have been mapped from hexadecimal code to color names. To see the available color names, you can either look at the `rgb.txt` file with a text editor, or you can use the `showrgb` command coupled with a paging program like `more` by executing the following command at the UNIX prompt:

```
showrgb | more
```

The `rgb.txt` file is usually located in the directory `/usr/lib/X11`. To move into this directory, enter the command:

```
cd /usr/lib/X11
```

Table 4.1 gives some examples of available colors in the `rgb.txt` file.

Table 4.1: *Some available colors in `rgb.txt`.*

violet	blue	green	yellow
orange	red	black	white
ghost white	peach puff	lavender blush	lemon chiffon
lawn green	chartreuse	olive drab	lime green
magenta	medium orchid	blue violet	purple

Hexadecimal Color Values

You can also specify a color by using a hexadecimal value from the Red, Green, and Blue (RGB) Color Model, common to most color displays. Each pixel on the screen is made up of three phosphors: one red, one green, and one blue. Varying the intensities of each of these phosphors varies the color that you see on your display.

You can specify the intensity of each of the three phosphors with a hexadecimal triad. The first part of the triad corresponds to the intensity of the red phosphor, the second to the intensity of the green phosphor, and the third to the intensity of the blue phosphor. A hexadecimal triad must begin with the symbol `#`. For example, the hexadecimal triad `#000` corresponds to no intensity in any of the

phosphors and yields the color black, while the triad #FFF corresponds to maximum intensity in all of the phosphors and yields white.

A hexadecimal triad with only one digit per phosphor allows for 4,096 (16^3) colors. Most displays are capable of many more colors than this, so you can use more than one digit per phosphor. Table 4.2 shows the allowed forms for an RGB triad; Table 4.3 illustrates hexadecimal values for some common colors. You can use up to four digits to specify the intensity of one phosphor (this allows for about 3×10^{14} colors). You do not need to know how many colors your machine can display; your window system automatically scales the color specifications to your hardware.

Table 4.2: *Legal forms of RGB triads.*

Triad Form	Approximate Number of Possible Colors
#RGB	4,000
#RRGGBB	17 million
#RRRGGBBB	70 billion
#RRRRGGGBBBB	3×10^{14}

Table 4.3: *Hexadecimal values of some common colors.*

Hex Value	Color Name
#000000	black
#FFFFFF	white
#FF0000	red
#00FF00	green
#0000FF	blue
#FFFF00	yellow

Table 4.3: Hexadecimal values of some common colors.

Hex Value	Color Name
#00FFFF	cyan
#FF00FF	magenta
#ADD8E6	light blue

Specifying Color Schemes

The following conventions are used when listing colors to specify a color scheme:

- Color names or values are separated by spaces.
- When a color name is more than one word, it should be enclosed in quotes. For example, “lawn green”.
- The order in which you list the color names or values corresponds to the numerical order in which they are referred to in Spotfire S+ with the graphics parameter `col`. For example, if you use the argument `col=3` in a S-PLUS plotting function, you are referring to the third color listed in the current color scheme.

Note

When specifying a color scheme in your X resources, the first color listed is the background color and corresponds to `col=0`.

- Colors are repeated cyclically, starting with color 1 (which corresponds to `col=1`). For example, if the current color scheme includes three colors (not including the background color), and you use the argument `col=5` in a S-PLUS plotting function, then the second color is used.
- You may abbreviate a list of colors with the specification `color1 n color2`. This list is composed of $(n+2)$ colors: `color1`, `color2`, and n colors that range smoothly between `color1` and `color2`. For example, the color scheme `blue red 10 "lawn`

green" specifies a list of 13 colors: blue, then red, then 10 colors ranging in between red and lawn green, and then lawn green.

Note

This method of specification is especially useful with the `image` plotting function.

- You may specify a list of colors as *halftones* with the specification `color1 hn color2`. This list is composed of $(n+2)$ "colors," which are actually tile patterns with progressively more `color2` on a background of `color1`. Halftone specifications are useful on devices with a limited number of simultaneous colors. For example, the color scheme `blue red h10 "lawn green"` specifies a list of 13 colors, just as our previous example did. In this example, however, only 3 entries in the X server's color table are allocated, rather than the 13 allocated by the previous example.

USING JAVA.GRAPH WINDOWS

The `java.graph` device is available only with Java-enabled versions of Spotfire S+. Using `java.graph`, you can change interactively the color specifications of your plots, and then immediately see the result. Also, you can change interactively the specifications that are used to send plots to a printer.

Starting and Stopping the `java.graph` Device

To start a `java.graph` graphics device, enter the following at the Spotfire S+ command prompt:

```
> java.graph()
```

The `java.graph` device is started automatically in the Java GUI version of Spotfire S+ if no other graphics device is open when you use Spotfire S+ to evaluate a high-level plotting function.

Note
The Java GUI is deprecated as of Spotfire S+ 8.1.

To stop the `java.graph` device, and close the graphics window without quitting Spotfire S+, use the `dev.off` or `graphics.off` function. Also, the `java.graph` device correctly shuts down if you close it using the standard window system method for closing a window.

An Example Plot

As you explore the various `java.graph` features, you can use the following S-PLUS code to generate an easily-reproducible graphic:

```
plot(corn.rain, corn.yield, type="n",
     main="Plot Example")
points(corn.rain, corn.yield, pch="*", col=2)
lines(lowess(corn.rain, corn.yield), lty=2, col=3)
legend(12, 23, c("Color 1", "Color 2", "Color 3"),
      pch=" * ", lty=c(1, 0, 2), col=c(1, 2, 3))
```

Note that in the call to `legend`, there is a space before and after the `*` in the argument `pch=" * "`.

To create the example plot in a `java.graph` window, first start the `java.graph` device:

```
> java.graph()
```

Then run the example code to create the plot shown in Figure 4.6.

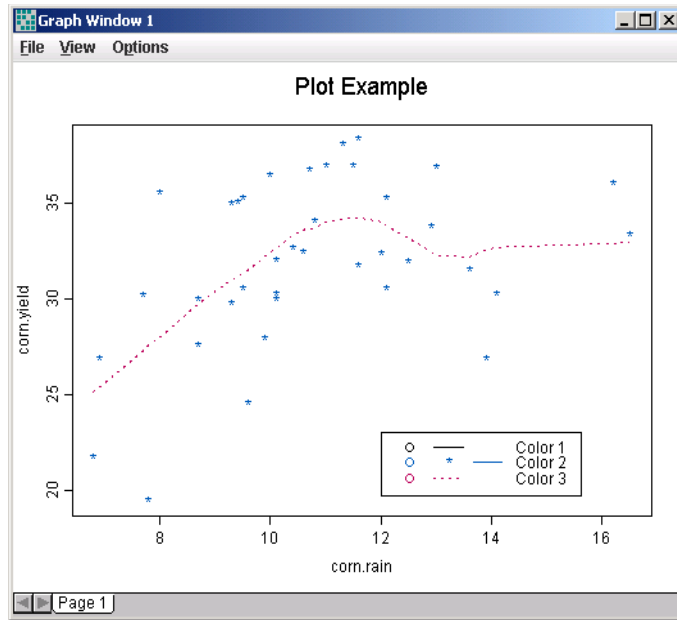


Figure 4.6: Plot example in a *java.graph* window.

By default, the color of the title, legend box, axis lines, axis labels, and axis titles are color 1. We have specified the points to have color 2, and the dashed line representing the smooth from the `lowess` command to have color 3.

Note

Figure 4.6 shows how the `java.graph` window appears when you start the `java.graph` device from the Java-enabled command-line version of Spotfire S+. If you start the `java.graph` device from the Spotfire S+ Java GUI instead, the window does not include the **File**, **View**, and **Options** menus; those menus are available in the Java GUI main window.

java.graph Window Features

The elements of the java.graph window are as follows:

- *Title bar* Contains a title of the form **Graph Window n**, the **Minimize** button, the **Maximize** button, the **Close** window button, and a window menu.
- *Page* Contains Spotfire S+ graphs that you create while the java.graph graphics device is active. A java.graph device can have multiple pages.
- *Resize Borders* Used to change the size of the window.
- *Tab bar* Displays the page tabs. Use it to quickly move between pages.

You can right-click the **Tab** bar to display a menu with the following options:

- *Zoom In* Expands the graph.
- *Zoom Out* Shrinks the graph.
- *Zoom to Rectangle* Expands the graph so the contents of a specified rectangle fills the window. Specify the rectangle by left-clicking in a corner, dragging the mouse, and then releasing it in the opposite diagonal corner. You must define the rectangle *before* choosing **Zoom to Rectangle** for the graph to be properly resized.
- *Fit in Window* Resizes the graph so that it fits completely within its window.
- *Set Graph Colors* Opens the **Set Graph Colors** dialog. This dialog is discussed in detail later in this section.
- *Graph Options* Opens the **Graph Options** dialog. This dialog is discussed in detail later in this section.
- *Page Properties...* Opens the **Page Properties** dialog, which allows you to specify a page title and page tag. To use this dialog, right-

- *Insert Page* click on the tab of the page that you want to modify; if you select **Page Properties** after simply right-clicking on the **Tab** bar, no dialog appears. Inserts a new page after the selected tab. To use this option, right-click on the tab of the page that should precede the new page, and choose **Insert Page**. If this tab is the currently active one, the new page is made active.
- *Delete Page* Deletes the selected tab and its associated page. To use this option, right-click on the tab of the page that should be deleted, and choose **Delete Page**.
- *Clear Page* Clears the selected page. To use this option, right-click on the tab of the page that should be cleared, and choose **Clear Page**.
- *Delete All Pages* Deletes all pages in the current graphics window.

Note that if you resize a `java.graph` window, the graph region resizes but maintains the same height-to-width ratio, adding gray borders on the sides if necessary. Printing a graph from a `java.graph` window also maintains the aspect ratio, expanding as much as possible to fill the page.

The Options Menu

If you are running `java.graph` in the Spotfire S+ Java GUI, the main **Options** menu contains options specific to the `java.graph` device. If you run `java.graph` in the Java-enabled command-line version of Spotfire S+, the **Options** menu in the graphics window is used to set options used by all `java.graph` devices.

Note

The information in the following “Set Graph Colors Dialog Box” section applies *only* if you are using Spotfire S+ in the backward compatible `use.device.palette(T)` mode. When using Spotfire S+ in `use.device.palette(F)` mode (which is the default), disregard this information and refer to the section Color Specification on page 3 for information about working with colors.

For information about backward compatibility for the Spotfire S+ graphics features, see section Backward Compatibility (page 24).

Set Graph Colors Dialog Box

Use the **Set Graph Colors** dialog to set the color scheme for the active `java.graph` window. The **Set Graph Colors** dialog (Figure 4.7) allows you change the colors in your plot interactively and immediately see the results.

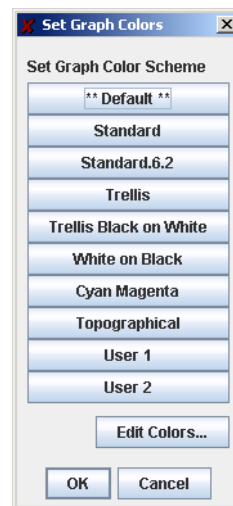


Figure 4.7: *The java.graph Set Graph Colors dialog box.*

The **Set Graph Colors** dialog box contains:

- Selection buttons for each of the available color schemes
- An **Edit Colors** button
- An **OK** button
- A **Cancel** button
- A **Help** button (available in the Java GUI only)

The Help Button

Click the **Help** button to view the help window for this dialog box, which contains essentially the information presented here.

Available Color Schemes

The following color schemes can be selected from the **Set Graph Colors** dialog:

- *Default* Used when graphs are first created in a `java.graph` window. Initially, the **Default** color scheme is the **Standard** color scheme, which uses a white background with a palette of darker colors for lines. However, you can customize this so that any color scheme appears by default in your graphs.
- *Standard* White background with a palette of darker colors for lines. Initially, this is used as the **Default** color scheme; it is available mainly so you can recover the initial **Default** color scheme after temporarily customizing it for your graphics.
- *Trellis* Gray background, mostly pastel line colors, and the cyan-magenta color scale for images.
- *Trellis Black on White* White background, various shades of gray for lines, and a grayscale for images.
- *White on Black* Grayscale color scheme with a black background, white and various shades of gray for lines, and a grayscale for images.
- *Cyan Magenta* White background, an assortment of line colors, and a cyan-magenta color scale for images. Unlike the other

cyan-magenta color scales, this one scales through black rather than through white.

- *Topographical* Similar to Cyan Magenta, except with image colors chosen to provide a reasonable representation of topographical data.
- *User 1, User 2* Similar to the standard color scheme, these are intended for further customization by end users.

Changing the Color Scheme

To select a different color scheme for your plot, click a name in the **Set Graph Colors** dialog. The name of the newly chosen color scheme is highlighted, and the selected **java.graph** window shows the chosen color scheme. This, however, is temporary. To make the change permanent, you must click on the **OK** button. If you click **Cancel**, the previous color scheme is restored.

Editing Colors

Each color scheme consists of four editable parts: a *name*, a *background color*, a set of *line colors*, and a set of *image colors*. To view the colors in a color scheme, click on **Edit Colors** in the **Set Graph Colors** dialog to display the **Edit Graph Colors** dialog shown in Figure 4.8.

Use the top of the **Edit Graph Colors** dialog to edit individual colors within a color scheme. To edit the background color, click the **Edit Background Color** button in the **Edit Graph Colors** dialog. To edit colors in the **Line Colors** or **Image Colors** palettes, click on a color rectangle, then select either the **Edit Selected Line Color** button or **Edit Selected Image Color** button as appropriate. The currently selected color is surrounded by a red border. You can select multiple consecutive colors by dragging the mouse over the desired colors; the red border appears around all selected colors.

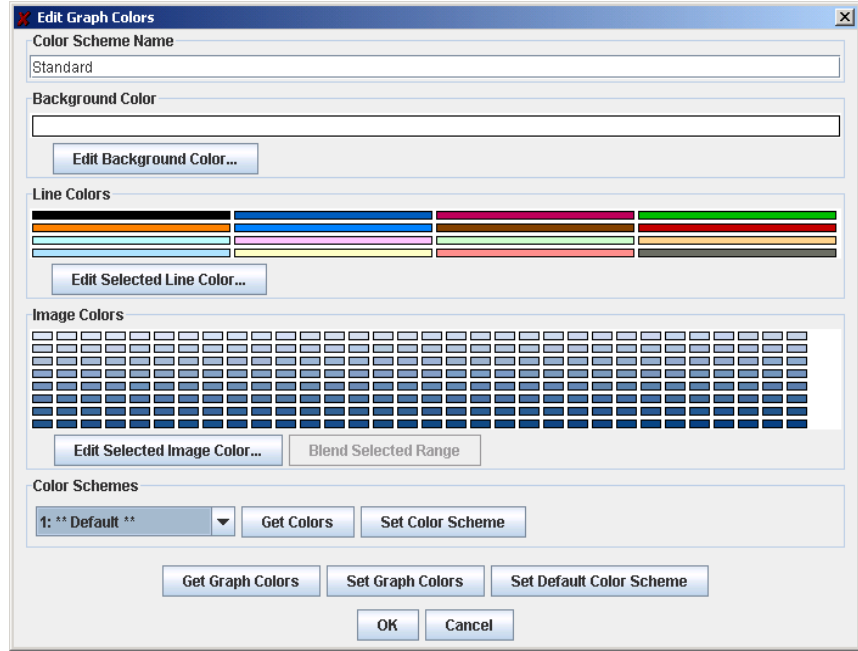


Figure 4.8: The *Edit Graph Colors* dialog.

The three buttons labeled **Edit xxx Color** in the **Edit Graph Colors** dialog bring up identical dialogs, titled **Edit xxx Color**. The **Edit Image Color** dialog is shown in Figure 4.9.

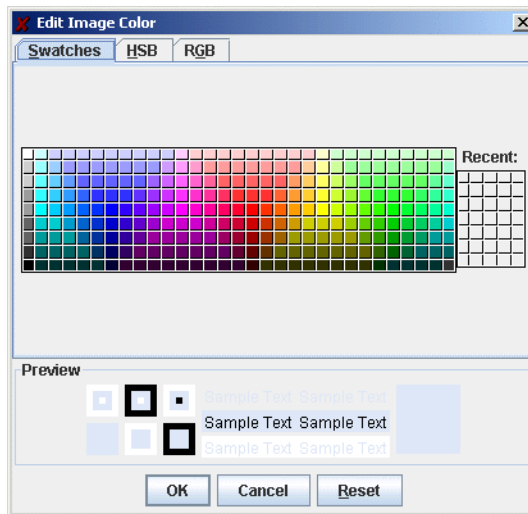


Figure 4.9: The *Edit Image Color* dialog.

Each dialog has three tabs: **Swatches**, **HSB**, and **RGB**. The three tabs provide alternative but equivalent methods for modifying your colors. The **Swatches** tab is the easiest to use: simply select a color from the palette of colors, examine the **Preview** section to see if it has the effect you're looking for, then click **OK**.

The **HSB** tab lets you specify colors using the HSB model (Hue-Saturation-Brightness) used by the PostScript page description language. Use this tab if you have an HSB color map you are trying to match exactly in your **java.graph** device. You can either specify the HSB values exactly, using the H, S, and B text fields, or relatively, by using the pointer on the color bar. The H values are drawn from a color wheel, so H accepts the values 0 to 359. The S and B values are percentages with 0 being none of the quality and 100 being full value. The color bar can select values for any of the three qualities, depending on which of the H, S, and B radio buttons is active. The H color bar appears as a rainbow of colors. The S color bar is the selected color shown with varying saturation, from white (no saturation) to full intensity color. The B color bar shows the amount of light in the color, from none (black) to full. The **HSB** tab also shows you, for your information only, the associated RGB color of the current HSB settings.

The **RGB** tab allows you to specify colors using the standard Red-Green-Blue color model. Use the sliders or the text fields to describe the appropriate RGB values.

Use the bottom of the **Edit Graph Colors** dialog to manipulate color schemes and graph colors, as follows:

- *Color Schemes popup list* Use this list to select one of the known color schemes. Note that selecting a color scheme does not update the colors in the **Edit Graph Colors** dialog.
- *Get Colors* Retrieves the colors from the color scheme selected in the popup list and update the displayed colors.

- *Set Color Scheme* Sets the color scheme selected in the popup list to the displayed colors. This setting is temporary until you click **OK**. If you click **Cancel**, the previous colors are restored.
- *Get Graph Colors* Retrieves the colors from the color scheme of the selected graph. This essentially restores the initial colors in the dialog, since the colors from the selected graph's color scheme are shown when the dialog first opens.
- *Set Graph Colors* Sets the color scheme of the selected graph window to be the current palette of colors. You can use this option to temporarily test combinations of colors on an active graph. To commit color changes made with this option, click the **OK** button; if you click **Cancel**, all changes are lost.
- *Set Default Color Scheme* Sets the default color scheme to the displayed colors. This is equivalent to selecting the default color scheme in the **Color Schemes** popup list, then clicking **Set Color Scheme**.

The Graph Options Dialog

The second graph menu item under the **Options** menu is labeled **Graph Options**. This brings up the **Graph Options** dialog, shown in Figure 4.10. Use the radio buttons under **New Plot Action** as described below to specify how the graphics window should respond to *clear commands*. Clear commands are generated whenever Spotfire S+ attempts to create a new high-level graphic.

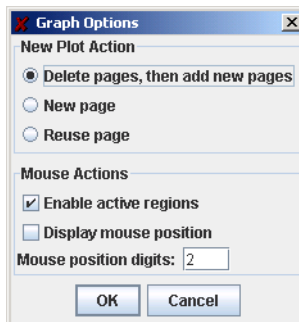


Figure 4.10: *The **Graph Options** dialog.*

- *Delete pages, then add new pages*

The first time that a clear command is issued to a **java.graph** device within a top-level expression, all existing pages in the window are deleted and a new “Page 1” is created. Additional clear commands within the top-level expression create additional pages. In this mode, graphics exist in the device only until a new top-level graphics expression replaces them.

- *New page*

Whenever a clear command is issued, create a new page. Use this mode to keep all your graphics for a session within a single **java.graph** device.

- *Reuse page*

Whenever a clear command is issued, clear the current page. In this mode, functions that display multiple plots will end up displaying just the last one.

Use the check boxes under **Mouse Actions** as follows:

- *Enable active regions*

Select this checkbox to enable active regions created with **java.identify** to be highlighted as the mouse passes over them and their associated actions to be performed when the mouse is clicked in the region. The default is selected.

- *Display mouse position*

Select this checkbox to display x - y coordinates of the mouse in the upper-right corner of the graph window. The text field immediately following, labeled **Mouse position digits**, allows you to specify the number of decimal digits to use when displaying mouse coordinates.

PRINTING YOUR GRAPHICS

Spotfire S+ produces camera-ready graphics plots for technical reports and papers. Spotfire S+ supports two kinds of hard copy graphics devices: PostScript laser printers and Hewlett-Packard HP-GL plotters. Spotfire S+ also supports publication on the World Wide Web by means of a graphics device for creating files in Portable Document Format (PDF), and popular word processing software by means of a graphics device for creating files in Enhanced Metafile Metafile (EMF) or Windows Metafile Format (WMF) and the ability of the `java.graph` graphics device to create popular bitmap formats. These devices are discussed in the following sections. General rules for making plot files are discussed in the section *Managing Files from Hard Copy Graphics Devices* (page 246).

Printing with PostScript Printers

For many Spotfire S+ users, producing graphics suitable for printing on PostScript-compatible printers is essential.

In Spotfire S+, you can create PostScript graphics using any of the following methods:

- Choose **Print** from the **Graph** menu on the `motif` windowing graphics device.
- Use the `printgraph` function with any graphics device that supports it. (The `motif` device supports `printgraph`, as do many others. See the `Devices` help file for a complete list.)
- Use the `postscript` function directly.

We discuss each of these methods in the following subsections.

If you are using `postscript` directly, the *aspect ratio* of the finished graphic is determined by the width and height, if any, that you specify, the orientation, and the paper size. If you use the other methods, by default the aspect ratio is the original aspect ratio of the device on which the graphic is originally created. For the windowing graphics device `motif`, this ratio is 8:6.32 by default. Resizing the graphics window has no effect on PostScript output created from the resized window; it retains the aspect ratio of the original, non-resized window.

Using the Print Option from the Motif Window

You can easily create PostScript versions of graphics created on the `motif` device by using the **Print** option from the **Graph** menu. The behavior of this option is determined by options specified in the **Printing Options** dialog box selected from the **Options** menu. The following choices are available:

- *Method* Verify that **PostScript** is selected.
- *Orientation* Determines the orientation of the graphic on the paper. **Landscape** orientation puts the x -axis along the long side of the paper; **Portrait** orientation puts the x -axis along the short side of the paper. To choose the orientation, move the pointer to the desired choice and click.
- *Command* A UNIX command executed when you select the **Print** option from the **Graph** menu. The default value, when **Method** is set to **PostScript**, is the command stored in the value of `ps.options()``$command`. To change this command, move the pointer to this line and click to ensure the line has input focus, then edit the command.

As the default command is normally to send a file to a printer, the most common use of the **Print** option is to create immediately a hard copy of the displayed graphic. You can, however, specify a command such as the following to store the PostScript output in a named file:

```
cat > myfile <
```

Here *myfile* is any desired file name. However, the `printgraph` function, described in the next section, provides a more convenient method for creating files of PostScript output.

To choose the **Print** option from the graphics device:

1. Move the pointer to the button labeled **Graph**.
2. Click and a menu appears.

3. Drag the pointer to the **Print** option, then release the mouse button. A message appears in the footer of the graphics window telling you that the specified command has been executed.

Print Options in the Java Graphics Window

You can print graphics created on the `java.graph` device by using the **Print** option from the main **File** menu. The **Print** dialog has the following options:

- *Copies* The number of copies of the graphic to print.
- *Print to* The name of a printer, or the file name to be used to print to a file.
- *Banner Page Title* The title to appear on the banner page of your print job, if your printer is configured to print a banner page.
- Print Command Options
 Additional options to be sent to your print command.

As the default command is normally to send a file to a printer, the most common use of the **Print** option is to create immediately a hard copy of the displayed graphic.

Using the printgraph Function

In its simplest use, the `printgraph` function is just another way to produce immediate hard copies of graphics created on windowing or other graphics devices. Many graphics devices for use with graphics terminals and emulators, including `tek14`, support the `printgraph` function.

The default behavior of the `printgraph` function is determined by a number of environment variables. These are discussed in the *User's Guide*. To make `printgraph` produce PostScript output, make sure the environment variable `S_PRINTGRAPH_METHOD` is set to `postscript`, or call `printgraph` directly with the argument `method="postscript"`.

`S_PRINTGRAPH_METHOD` determines the default value for the method argument to `printgraph` and specifies the type of printer for which `printgraph` produces output. Environment variables cannot be set from within Spotfire S+; if you want to change an environment variable, quit Spotfire S+, reset the environment variable, then restart Spotfire S+.

Within your Spotfire S+ session, you can control the default printing behavior by using `ps.options`. We recommend that you use `ps.options` instead of environment variables whenever possible. The options that can be controlled through `ps.options` are described in the section Setting PostScript Options (page 241).

To call `printgraph` to print an immediate hard copy of the current graphic, use the following call:

```
> printgraph()
```

You can override the default method, command, and orientation with arguments to `printgraph`:

```
> printgraph(horizontal=F, method="postscript",  
+ command="lpr -h")
```

Using the postscript Function

You can start the `postscript` device directly very simply as follows:

```
> postscript()
```

By default, this writes PostScript output to a temporary file using the template specified in `ps.options`. When the device is shut down, the output is printed with the command specified in `ps.options`.

You can specify many options as arguments to `postscript`; most of these are global PostScript printing options that are also used by the Print option of the windowing graphics device and by the `printgraph` function—these options are discussed in the section Setting PostScript Options (page 241). The `append`, `onfile`, and `print.it` arguments, however, are specific to calls to `postscript`.

The `onfile` argument is specified as a logical value, which defaults to `TRUE`. By default, when you start the `postscript` device explicitly, plots are accumulated into a single file as given by the `file` argument. If no `file` argument is specified, the file is named using the template specified in `ps.options()$tempfile`. When `onfile` is `FALSE`, a separate file is created for each plot and the PostScript file

created is structured as an Encapsulated PostScript document. See the section *Creating Encapsulated PostScript Files* (page 239), for further details.

The `append` option is a logical value that specifies whether PostScript output is appended to `file` if it already exists. In addition to appending the new graphics, Spotfire S+ edits the file to comply with the PostScript Document Structuring Conventions. If `append=FALSE`, new graphics output writes over the existing file, destroying its previous contents.

You can use the `print.it` argument to specify that the graphic created on the `postscript` device be both sent to the printer and written to a file, as follows:

```
> postscript(file="mystuff2.ps", print.it=T)
> plot(corn.rain)
> title("A plot created with postscript()")
> dev.off()
```

```
Starting to make postscript file.
```

```
  null device
        1
```

```
> !vi mystuff2.ps
```

```
%!PS-Adobe-3.0
%%Title: (Spotfire S+ Graphics)
%%Creator: Spotfire S+
%%For: (John Doe,x240)
%%CreationDate: Thur Sep 04 11:45:21 2008
%%BoundingBox: 20 11 592 781
%%Pages: (atend)
. . .
```

Warning

If you want to both print the graphic and keep the named PostScript file, be sure that the UNIX print command does not delete the printed file. For example, on some computers, the default value of `ps.options()$command` (which is determined by the environment variable `S_POSTSCRIPT_PRINT_COMMAND`) is `lpr -r -h`, where the `-r` flag causes the printed file to be deleted. The following call to `postscript` replaces this default with a command that does not delete the file:

```
> postscript(file="mystuff2.ps", print.it=T, command="lpr -h")
```

Using `postscript` directly can be cumbersome, since you don't get immediate feedback on graphics produced incrementally. You can, however, build a *graphics function* incrementally, using a windowing graphics device or graphics terminal. Then, when the graphics function works well on screen, start a `postscript` device and call your graphics function. Such an approach will result in fewer hard copies for the recycling bin. For example, consider the code below, which combines into a single function the commands needed for creating a complicated graphic:

```
> usasymb.plot
function()
{
  select <- c("Atlanta", "Atlantic City", "Bismarck",
             "Boise", "Dallas", "Denver", "Lincoln",
             "Los Angeles", "Miami", "Milwaukee",
             "New York", "Seattle")
  city.name <- city.name
  city.x <- city.x
  city.y <- city.y
  names(city.x) <- names(city.y) <-
    names(city.name) <- city.name
  pop <- c(425, 60, 28, 34, 904, 494, 129, 2967, 347,
          741, 7072, 557)
  usa()
  symbols(city.x[select], city.y[select], circles =
    sqrt(pop), add = T)
  size <- ifelse(pop > 1000, 2, 1)
  size <- ifelse(pop < 100, 0.5, size)
  text(city.x[select], city.y[select], city.name[
    select], cex = size)
}
```


Modifying a function containing a string of graphics commands is much easier than retyping all the commands to re-create the graphic.

Another useful technique for preparing PostScript graphics is to use PostScript screen viewers such as `ghostview`.

Creating Encapsulated PostScript Files

If you are creating graphics for inclusion in other documents, you typically want to create a single file for each graphic in a file format known as *Encapsulated PostScript*, or EPS. EPS files can be included in documents produced by many word-processing and text-formatting programs.

Documents conforming to the Adobe Document Structuring Convention Specifications, Version 3 for Encapsulated PostScript have the following first line:

```
%!PS-Adobe-3.0 EPSF-3.0
```

They must also include a `BoundingBox` comment. Non-EPS files have the following first line:

```
%!PS-Adobe-3.0
```

Warning

Spotfire S+ supports the Encapsulated PostScript file format, EPSF. It *does not* support the Encapsulated PostScript Interchange format, EPSI. EPS files created by Spotfire S+ do not include a preview image, so if you import a Spotfire S+ graphic into WYSIWYG software such as FrameMaker or Word, you will see only a gray rectangle or a box where the graphic is included.

You can use `printgraph` to produce separate files for each graphic you produce, as soon as you've finished composing it on a windowing graphics device or terminal/emulator that supports `printgraph`. You can specify the file name and orientation of the graphics file. For example, you can create the PostScript file `mystuff.ps` containing a plot of the data set `corn.rain` as follows:

```
> motif()
> plot(corn.rain)
> title("My Plot of Corn Rain Data")
> printgraph(file="mystuff.eps")
```

You can produce EPS files with direct calls to `postscript` by setting `onefile=FALSE`. To create a *single* file, with a name you specify, call `postscript` with the `file` argument and `onefile=F`:

```
> postscript(file="mystuff.eps", onefile = F, print = F)
> plot(corn.rain)
> dev.off()
```

Warning

If you supply the `file` argument and set `onefile=F` in the same call to `postscript`, you *must* turn off the device with `dev.off` after completing the first plot. Otherwise, the next plot will overwrite the previous plot, and the previous plot will be irretrievably lost.

To create a *series* of Encapsulated PostScript files in a single call to `postscript`, omit the `file` argument:

```
> postscript(onefile=F, print=F)
> plot(corn.rain)
> plot(corn.yield)
```

```
Starting to make postscript file.
Generated postscript file "ps.out.0001.ps".
```

Because `onefile` is `FALSE`, `postscript` generates a postscript file as soon as the new call to `plot` tells it that nothing more will be added to the first plot. The file `ps.out.0001.ps` contains the plot of `corn.rain`. A file containing the plot of `corn.yield` is generated as soon as a new call to `plot` or a call to `dev.off` closes the old plot.

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

```
Starting to make postscript file.
Generated postscript file "ps.out.0002.ps".
```

You can give a series-specific naming convention for the series of files using the `tempfile` argument to `postscript`:

```
> postscript(onefile=F, print=F, tempfile="corn.####.ps")
> plot(corn.rain)
> plot(corn.yield)
```

```
Starting to make postscript file.
Generated postscript file "corn.0001.ps".
```

```
> plot(corn.rain, corn.yield)
Starting to make postscript file.
Generated postscript file "corn.0002.ps".

> dev.off()
Starting to make postscript file.
Generated postscript file "corn.0003.ps".
```

Setting PostScript Options

The behavior of the `postscript` graphics device, whether activated by the Print option from a `motif` graphics device, by a call to `printgraph`, or by a direct call to `postscript`, is controlled by options you can set with the `ps.options` function. These options allow you to control many aspects of the PostScript output, including the following:

- The name of the PostScript output file.
- The UNIX command to print your PostScript output.
- The orientation and size of the finished plot.
- Printer-specific characteristics, including paper size, number of rasters per inch, and the size of the imageable region.
- Plotting characteristics of the graphics, including the base point size for text and available fonts and colors.

Specifying the PostScript File Name

All PostScript output is initially written to a file. Unless you explicitly call the `postscript` device with the `onefile=T` argument, Spotfire S+ writes a separate PostScript file for each plot, in compliance with the Encapsulated PostScript Document Structuring Conventions. You can specify the file name for the output file using the `file` argument to `postscript` or `printgraph`, or provide a template for multiple file names using the PostScript option `tempfile`, which defaults to `"ps.out.####.ps"`. You can specify this option as an argument to the `printgraph`, `postscript`, and `ps.options` functions. The template you specify must include some `#` symbols, as in the default. Spotfire S+ replaces the first series of these symbols that it encounters with a sequential number of the same number of digits in the generated file names. For example, if you have a project involving the halibut data, and you know your project will use fewer than 1000 graphics files, you can set the `tempfile` option as follows to use the name of your data set:

```
> ps.options(tempfile="halibut.###.ps")
```

Specifying a Printer Command

What happens to the file after it is created is determined by the `command` option. The `command` option is a character string specifying the UNIX command used to print a graphic. If `file` is specified (and is neither a template nor an empty string), the `command` option must be activated by some user action, either choosing the Print option from a windowing graphics device, specifying `print=TRUE` in the `printgraph` function, or specifying `print.it=TRUE` in the `postscript` function.

The default for `command` is the value of the environment variable `S_POSTSCRIPT_PRINT_COMMAND`.

Specifying Plot Orientation and Size

You specify the plot orientation with the `horizontal` option: `TRUE` for landscape mode (x -axis along long edge of paper), `FALSE` for portrait. Most figures embedded in documents should be created in portrait mode, because that is the usual orientation of documents. The default is the orientation specified by the `S_PRINT_ORIENTATION`, which by default is set to `TRUE`, that is, landscape mode. If you specify an orientation with your graphics window's Options Printing menu, that specified orientation is taken to be the default.

You specify the plotting region, in inches, with the `width` (the x -axis dimension) and `height` (y -axis dimension) options. Thus, to create graphics for inclusion in a manual, you might specify the following options:

```
> ps.options(horizontal=F, width=5, height=4)
```

The default value for `width` and `height` are determined by the printer's imageable region, as described in the next subsection.

Specifying Printer Characteristics

PostScript can describe pages of virtually any size, but it does little good to create enormous page descriptions if you don't have an output device capable of printing them. Most PostScript printers have remarkably similar characteristics, so you may not have to change the options that specify them. For example, in the United States, most printers default to "letter" (8 1/2 x 11) paper. Among the options that you can specify for your printer, the paper option is the most important. The paper argument is a character string; most standard ANSI and ISO paper sizes are accepted. Each paper size has a specific *imageable region*, which is the portion of the page on which the printer can actually print. This region can vary slightly depending on

the printer hardware, even for paper of the same size. The imageable region determines the default values for the width and height options.

Specifying Plotting Characteristics

The PostScript options that have the greatest immediate impact on what you see are those affecting the PostScript graphic's plotting characteristics. These options include the following:

- `fonts` A vector of character strings specifying all available fonts.
- `colors` A numeric vector or matrix assigning actual colors to the color numbers used as arguments to graphics functions. This option is discussed in more detail in the next section.
- `image.colors` Same as `colors`, but for use with the `image` function.
- `background` A numeric vector giving the color of the background, as in `colors.background`, can also be a single number that is used as an index to the `colors` argument if it is positive or, if it is negative, specifies no background at all.

Creating Color PostScript Graphics

Creating PostScript graphics in color is no more difficult than creating color graphics on your windowing graphics device. With the `xgetrgb` function, you can copy the color map from the current `motif` device and use it for PostScript output. The following steps show how to print graphics from a `motif` window to a PostScript printer using the same color map.

1. Start the graphics window:

```
> motif()
```
2. Set the color scheme using the **Color Scheme** dialog box, accessible from the **Options** menu.
3. Plot the graphic in the graphics window:

```
> image(voice.five)
```

4. Capture the colors from the device using `xgetrgb`:

```
> my.colors <- xgetrgb(type="images")
```

The `type` argument to `xgetrgb` should be appropriate for the type of graph being reproduced. Here, we use `type="images"` because we want the colors used to produce an image plot. The default type is `"polygons"`, which is appropriate for bar plots, histograms, and pie charts, and is usually also suitable for scatter plots and line plots such as time series plots. Other valid types are `"lines"`, `"text"`, and `"background"`.

5. Send the color specification to update the graphics window's printer options:

```
> ps.options.send(image.colors=my.colors)
```

The `image.colors` argument assigns colors for image plots. Use the `colors` argument to assign colors for all other plots. Use the `background` argument to specify the background color.

You can, of course, use the results of `xgetrgb` as arguments without first assigning them to a Spotfire S+ object, as is shown below:

```
> ps.options.send(image.colors=xgetrgb("images"),  
+ colors=xgetrgb("lines"),  
+ background = xgetrgb("background"))
```

6. Select the **Print** button to print the colored graphic.

To create color graphics with the `postscript` function, you follow essentially the same steps, as in the following example:

1. Start the graphics window:

```
> motif()
```

2. Set the desired color scheme using **Options ► Color Scheme** from the `motif` menu.
3. Capture the colors from the device using `xgetrgb` and specify the captured colors as the PostScript color scheme using `ps.options`:

```
> ps.options(colors = xgetrgb("lines"),  
+ background = xgetrgb("background"))
```

4. Start the postscript device using the `postscript` function:

```
> postscript(file = "colcorn.ps")
```

5. Plot the graphic; the following commands produce a plot with three different colors:

```
> plot(corn.rain, corn.yield, type="n")
> points(corn.rain, corn.yield, col=2)
> title(main="A plot with several colors", col=3)
```

6. Turn off the postscript device:

```
> dev.off()
```

Creating Bitmap Graphics

Bitmap graphics are popular because they are easy to include into most word processing software. They are not recommended for most statistical graphics, because they tend to have lower resolution than normal Spotfire S+ vector graphics, such as those produced on screen by the `java.graph` or `motif` devices, or in files by the `postscript`, `pdf.graph`, `emf.graph`, or `wmf.graph` devices. Bitmaps can be useful for image graphics, such as those produced by the `image` function.

To create a bitmap graphic, start `java.graph` with a `file` argument and, if necessary, a `format` argument. The supported format arguments are "JPEG", "BMP", "PNG", "PNM", "SPJ", and "TIFF".

Note

`java.graph` interprets the file type from the file extension if it is specified by `file`. If the file extension is not part of the file name, and you specify no format, `java.graph` defaults to JPEG. If the file extension contains an unsupported type, or if format specifies an unsupported type, `java.graph` defaults to JPEG.

To create a JPEG image of the `voice.five` data, use `java.graph` as follows:

```
java.graph("voice.jpeg")
image(voice.five)
dev.off()
```

Managing Files from Hard Copy Graphics Devices

With all hard copy graphics devices, a plot is sent to a plot file not when initially requested, but only after a subsequent high-level graphics command is issued, a new frame is started, the graphics device is turned off, or you quit Spotfire S+. To write the current plot to a plot file (assuming you have started the graphics device with the appropriate file option), you must do one of the following:

- Make another plot (assuming a single figure layout).
- Call the function `frame` (again, assuming a single figure layout).
- Call the function `dev.off` to turn off the current graphics device.
- Call the function `graphics.off` to turn off all of the active graphics devices.
- Quit Spotfire S+.

Once you have created a graphics file, you can send it to the printer or plotter without exiting Spotfire S+ by using the following procedure:

1. Type `!` to escape to UNIX.
2. Type the appropriate printing command, and then the name of the file.
3. Type a carriage return.

To remove graphics files after sending them to the plotter without exiting Spotfire S+:

1. Type `!` to escape to UNIX.
2. Type `rm file`, where *file* is the name of the graphics file you want removed.
3. Type a carriage return.

Using Graphics from a Function or Script

Most experienced users of Spotfire S+ use a function or *script* to construct complicated plots for presentation or publication. This method lets you use the `motif` display device to preview the plots on

your screen, and then, once you are satisfied with your plots, send them to a hard copy device without having to re-type the same plotting commands.

Note

Direct use of a hard copy device ensures the best hard copy output.

To use this method using a S-PLUS function, follow these steps:

1. Put all the Spotfire S+ commands necessary to create the graphs into a function in S-PLUS (say `plotfcn`) using `fix`. Do *not* include commands that start a graphics device.
2. In Spotfire S+, start a graphics device, then call your function:

```
> motif()
> plotfcn()
```

Note

If you are creating several plots on separate pages, you may want to set the graphics parameter `ask` to `TRUE` before calling your plotting function. In this case, the sequence of steps is:

```
> motif()
> par(ask = T)
> plotfcn()
```

3. View your graphs. If you want to change something, use `fix` to modify your plotting function.
4. Once you are satisfied with your plots, start a hard copy graphics device, call your function, and then turn the hard copy graphics device off:

```
> postscript()
> plotfcn()
> dev.off()
```

5. Save your function containing graphics commands if you will need to reproduce the plots in the future.

To use this method using a script, follow these steps:

1. Put all the Spotfire S+ commands necessary to create the graphs into a file outside of Spotfire S+ (say `plotcmds.asc`) using an editor (e.g., `vi`). Do *not* include commands that start a graphics device.
2. In Spotfire S+, start a graphics device, then use `source` to execute the Spotfire S+ commands in your file:

```
> motif()  
> source("plotcmds.asc")
```

3. View your graphs. If you want to change something, edit your file with an editor.
4. Once you are satisfied with your plots, start a hard copy graphics device, source your plotting commands, and then turn the hard copy graphics device off:

```
> postscript()  
> source("plotcmds.asc")  
> dev.off()
```

5. Save your file of graphics commands if you will need to reproduce the plots in the future.

EDITING GRAPHICS IN WINDOWS

5

Graphs	250
The Graph Sheet	251
Methods for Creating a Graph	252
Changing the Plot Type	253
Adding a Plot to a Graph	254
Placing Multiple Graphs on a Graph Sheet	256
Projecting a 2D Plot Onto a 3D Plane	258
Trellis Graphics	260
Formatting a Graph	264
Formatting a Graph: An Example	266
Formatting a Graph Sheet	271
Formatting the Graph	272
Formatting 2D Axes	275
Formatting 2D Axis Labels	277
Adding and Formatting Multiline Text	278
Adding Titles and Legends	280
Adding Labels for Points	282
Adding a Curve Fit Equation	283
Adding Lines, Shapes, and Symbols	284
Modifying Image Colors	284
Working With Graph Objects	286
Plot Types	288
Formatting a Graph (Continued)	288
Using Graph Styles and Customizing Colors	291
Embedding and Extracting Data in Graph Sheets	293
Linking and Embedding Objects	294
Data From Another Application	294
Embedding Spotfire S+ Graphics in Other Applications	295
Printing a Graph	297
Exporting a Graph to a File	298

GRAPHS

The ability to create graphs quickly and easily is one of the most powerful tools in Spotfire S+. If you have not already done so, it would be a good idea to go through the tutorial booklet *Getting Started with Spotfire S+* before continuing to read this chapter. Spotfire S+ can generate a wide variety of 2D and 3D plots, and we will concentrate on only a few of these in this chapter. For information on the other plots and for much more detail, see the Chapter 4, Creating Plots in the *Spotfire S+ User's Guide for Windows*, and the online help.

Note

If your plot was created from the command line, as described in Chapter 2, Traditional Graphics or Chapter 3, Traditional Trellis Graphics and you want to edit it using the GUI, you must first convert the plot to graphic objects as follows:

1. At the command prompt, enter `use.legacy.graphics(T)` to put the Spotfire S+ graphics system into legacy mode.
2. Right-click inside the plot and select **Convert to Objects** from the shortcut menu.

Reading This Chapter

Don't read this chapter! That is, do not read it from beginning to end. Much of the material describes procedures for doing a certain action, such as changing the plot type or formatting an axis. Unless you need to perform such a task, there is no need to read the procedure.

To understand how to create graphs, we suggest the following steps:

1. Read the booklet *Getting Started with Spotfire S+* for the basics on how to create graphs and look through the example plot types.
2. Read the information on importing, exporting, and exploring data in the *Spotfire S+ User's Guide*.
3. Read this section for basic terminology and graph creation information.
4. Skim the remainder of this chapter and refer back to the material as needed.

This should provide you with a good overview of the graphics capabilities of Spotfire S+.

The Graph Sheet

In Spotfire S+ we distinguish between the **Graph Sheet**, the graph area, and the plot area. The **Graph Sheet** is best described as the sheet of paper on which we draw our plots. When we print, we print one or more pages of the **Graph Sheet**. A **Graph Sheet** can contain more than one graph. The graph area refers to the rectangle surrounding the data points, axes, legends, graph title, etc. The plot area is the rectangular area within the graph where the data are plotted. See Figure 5.1 for an illustration.

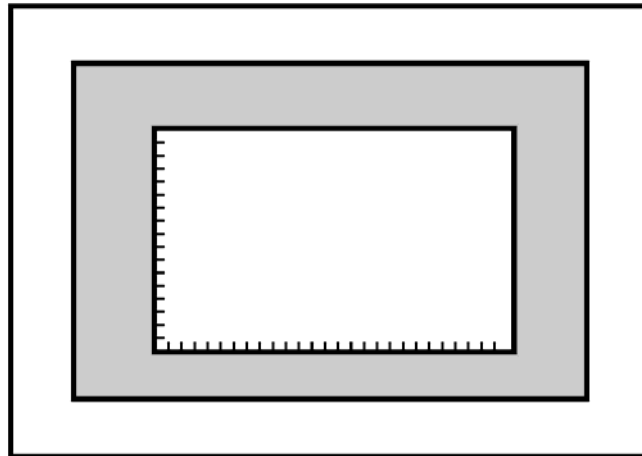






Figure 5.1: A **Graph Sheet** with graph area (gray) and plot area (center).

Creating, Opening, Saving, and Printing

You can create a new **Graph Sheet** using the **New** button  on the **Standard** toolbar. To save or print a **Graph Sheet**, you must first select the **Graph Sheet** by clicking it. Then use the **Save** button  to save the **Graph Sheet** or the **Print** button  to print it to your printer. To open a previously saved **Graph Sheet**, use the **Open** button  (**Graph Sheets** have **.sgr** file extensions). These **Graph Sheet** functions can also be accessed from the **File** menu.

Sometimes you may want to export your Spotfire S+ graph into an image file format, such as a Windows bitmap (.BMP), JPEG (.JPG), or Paintbrush (.PCX). To do this, choose **File ► Export Graph** from the main menu.

Viewing Graph Sheets



You can zoom your **Graph Sheets** to focus on a particular area by choosing **View ► Zoom**. Press F2 to view the graph at full screen without the menu bar, window title bar, or toolbars. Click the mouse or any keyboard button to return to the original view. Note that **Graph Sheets** are always printed at 100% size, even if they are zoomed.

To increase the display speed of **Graph Sheets**, you can use Draft mode. You can toggle this option on and off by choosing **View ► Draft**. This option is helpful for users with slow computer systems. Draft mode does not affect the print quality of your plots.

Methods for Creating a Graph



There are several methods for creating graphs. You can select data and click a plot palette button, you can drag and drop plot buttons onto graphs and then drag data onto the plot buttons, or you can use the **Graph** option on the **Insert** menu. The first two methods are described below.

Creating a Graph Using Plot Buttons

The **2D Plots** button  and **3D Plots** button  are available on the **Standard** toolbar for creating graphs quickly. When you click the **2D** or **3D Plots** button, a palette of plot buttons appears. For a description of each plot, move the mouse cursor over each button in the palette. A text description of the plot type appears.

When a new graph is created using a plot button, a new **Graph Sheet** is automatically opened.

Creating a graph using a plot button



1. Click the **2D Plots** button  or **3D Plots** button  on the **Standard** toolbar to open a palette of available plot types.
2. Open the **Data** window or **Object Explorer** containing the data to plot.
3. Select the data columns you want to plot. Use CTRL-click to select noncontiguous columns. The order in which columns are selected determines their default plotting order.

4. Click the desired plot button on the palette. If you hold the mouse over a plot button, a description of the plot type appears.
5. A new **Graph Sheet** opens, and the graph is drawn on the **Graph Sheet**.

Creating a Graph Using Drag-and-Drop

You can also create a graph by dragging and dropping each component of a graph onto a **Graph Sheet**.

Creating a graph using drag-and-drop



1. Open the **Data** window or **Object Explorer** containing the data to plot.
2. Create a new **Graph Sheet** or open an existing **Graph Sheet**.
3. From the main menu, choose **Window ► Tile Vertical**. Now you can see the data and the **Graph Sheet** simultaneously. Select the **Graph Sheet** window by clicking its title bar.
4. Click the **2D Plots** button  or **3D Plots** button  on the **Standard** toolbar. A palette of available plot types appears.
5. Drag the desired plot button from the palette and drop it inside the **Graph Sheet**. Default axes are drawn, and a plot icon is drawn on the graph.
6. Select the data columns you want to plot. Use CTRL-click to select noncontiguous columns.
7. Position the mouse within the selected region (not in the column header) until the cursor changes to an arrow. Drag the data with the mouse and move it over a plot icon. When the plot icon changes color, release the mouse button to drop the data and generate the plot.
8. The plot icon is replaced by an actual plot of the data columns.

Changing the Plot Type

Once a graph is created, it is possible to change the plot type. For example, suppose you have created a scatter plot and now you want to create a linear fit plot with the same data. By following the above

procedures for creating a new plot, you can create the linear fit plot. Instead of creating a new plot, you can also change the plot type of the existing plot.

Changing the plot type using a plot palette

1. Select the plot you want to change. A green knob appears on the data point closest to the x -axis when the plot is properly selected.
2. Click the **2D Plots** button  or **3D Plots** button  on the **Standard** toolbar. A palette of available plot types appears.
3. Click the desired plot button. The selected plot is redrawn using the new type. You can cycle through multiple plot types by clicking on the appropriate plot buttons.

The only caveat is that the plot types you select must have the same data requirements. For example, you can change a scatter plot into a linear fit plot since both require the same type of data, but you cannot change a 2D scatter plot into a 3D scatter plot because a 2D scatter plot only requires two columns of data, while a 3D scatter plot requires three columns of data. If your data are not appropriate for the chosen plot type, the plot appears on the graph in an iconized form. An alternative way of changing the plot type is by selecting the plot and then choosing **Format ► Change Plot Type** from the main menu.

Adding a Plot to a Graph

Plots can be added to an existing graph using the plot buttons or the menus.

Each plot on a graph represents one or more data columns. The plots can all be of the same plot type (for example, line plots) or a combination of plot types (for example, line, scatter, and bar plots).

Combined plots must have the same *type* of axes. For example, both a line plot and a bar chart have xy axes and can be combined on one graph. However, a boxplot and a surface plot cannot be combined on the same graph because they have different types of axes. A 2D graph and a 3D graph *can* both be placed on the same **Graph Sheet**, but they will not be on the same graph.

You can easily add plots to an existing graph by selecting the graph, selecting the data, and SHIFT-clicking the plot buttons.

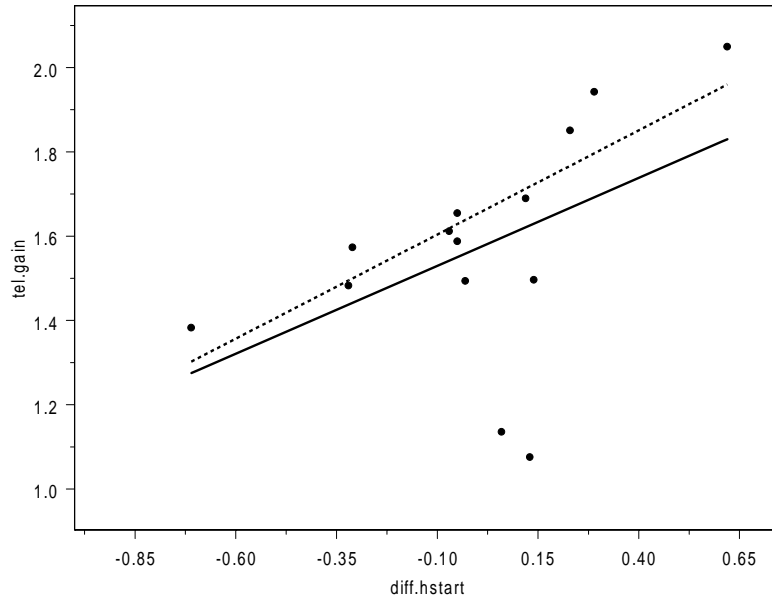




Figure 5.2: *Multiple plots on a single graph.*

Adding a plot using a plot button

1. Select the graph to which you want to add the plot.
2. Open the **Data** window containing the data to plot or select the data in the **Object Explorer** so that the columns appear in the right pane.
3. From the main menu, choose **Window ► Tile Vertical**. Now you can see the data and the **Graph Sheet** simultaneously.
4. Select the data columns you want to plot. Use CTRL-click to select noncontiguous columns.
5. Click the **2D Plots** button  or **3D Plots** button  on the **Standard** toolbar. A palette of available plot types appears.
6. SHIFT-click the desired plot button on the palette.

The plot is added to the selected graph using the selected data columns. If no graph is selected before SHIFT-clicking, a *new* graph will be added to the current **Graph Sheet**.

Note

Use the following general approach any time you want to add another graph line or curve fit to a scatter plot:

1. Select the graph region.
2. Select the data.
3. Press the SHIFT key and click the plot palette button for the line or curve you want to add to your scatter plot.

**Placing
Multiple
Graphs on a
Graph Sheet**

Graphs can be added to an existing **Graph Sheet** using the plot buttons, the menus, or drag-and-drop. Plots can be added either to the current page, or you can create new pages to hold additional graphs.

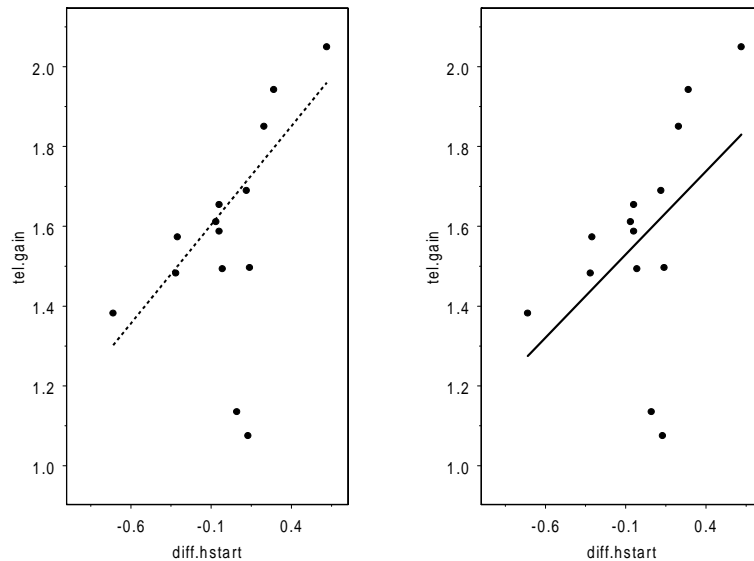




Figure 5.3: *Multiple graphs on a page.*



Adding a graph by SHIFT-clicking on a plot button

1. Open the **Graph Sheet** in which you want to add a graph. Make sure nothing on the **Graph Sheet** is selected. If a graph is selected, this procedure will add a plot to the selected graph instead of adding a new graph.

2. Open the **Data** window containing the data to plot or view the data in the **Object Explorer**.
3. From the main menu, choose **Window ► Tile Vertical**. Now you can see the data and the **Graph Sheet** simultaneously.
4. In the **Data** window, select the data columns you want to plot. Use CTRL-click to select noncontiguous columns.
5. Click the **2D Plots** button  or **3D Plots** button  on the **Standard** toolbar. A palette of available plot types appears.
6. SHIFT-click the desired plot button on the palette.

The graph is added to the current **Graph Sheet**, and a plot is placed on the graph using the selected data columns.

Adding a graph using drag-and-drop

1. Open the **Data** window containing the data to plot or view the data in the **Object Explorer**.
2. Open the **Graph Sheet** in which you want to add a graph.
3. From the main menu, choose **Window ► Tile Vertical**. Now you can see the data and the **Graph Sheet** simultaneously. Select the **Graph Sheet** window by clicking its title bar.
4. Click the **2D Plots** button  or **3D Plots** button  on the **Standard** toolbar. A palette of available plot types appears.
5. Drag the desired plot button from the palette and drop it inside the **Graph Sheet**. Default axes are drawn, and a plot icon is drawn on the graph.
6. Select the data columns you want to plot. Use CTRL-click to select noncontiguous columns.
7. Position the mouse within the selected columns until the cursor changes into an arrow. Pressing the left mouse button, drag the data and move it over a plot icon. When the plot icon changes color, release the mouse button to drop the data and generate the plot.

Adding a new page to a Graph Sheet

1. Open the **Graph Sheet** in which you want to add a graph.
2. Right-click the page tab at the bottom of the **Graph Sheet** and select **Insert Page** from the shortcut menu.
3. Add a plot to the new page as described above using either SHIFT-click or drag-and-drop.

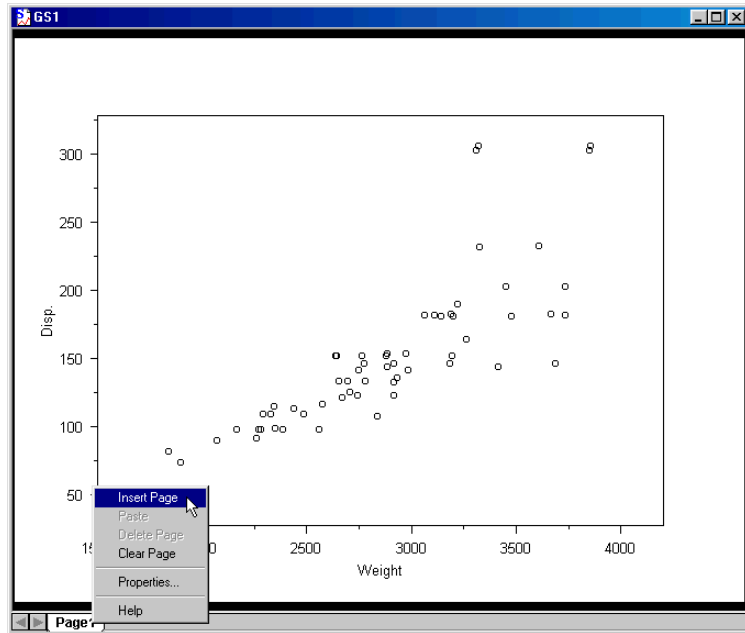


Figure 5.4: Adding a page to a *Graph Sheet*.

Projecting a 2D Plot Onto a 3D Plane

Most of the 2D plot types can be projected onto a 3D plane. This can be useful if you want to combine multiple 2D plots in 3D space and then rotate the results. You can drag-and-drop a 2D plot button onto a 3D plane, or you can select **Graph** from the **Insert** menu to create a projected 2D plot.

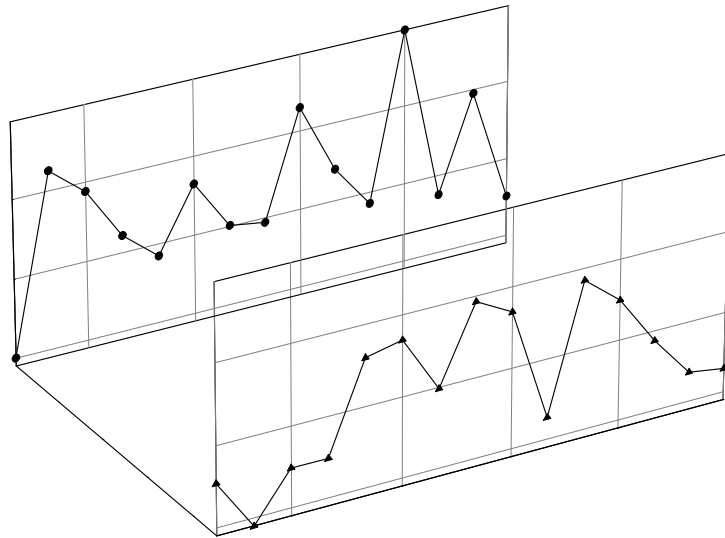




Figure 5.5: *Multiple 2D plots in 3D space.*

Projecting a 2D plot using drag-and-drop

1. Create a new **Graph Sheet**.
2. Click the **3D Plots** button  on the **Standard** toolbar to display the plot palette.
3. There are six 3D plane combinations on the **Plots 3D** palette.



Drag one of the 3D plane buttons off the plot palette and drop it onto the **Graph Sheet**. A 3D graph is drawn, and the plane is added automatically to the graph. The plane is automatically positioned at the minimum or maximum position, depending on which plane button you choose. You can drag and drop additional 3D planes as desired.

4. Click the **2D Plots** button  on the **Standard** toolbar to display the plot palette.
5. Drag-and-drop a 2D plot button onto the desired 3D plane. As you drag the plot over a 3D plane, the plane becomes highlighted (because it is an active drop target).

6. The plot icon is now linked to the 3D plane. You can double-click the plot icon to specify your data columns, or you can drag and drop data columns directly from your data.

When you have specified the data, the 2D plot is drawn on the specified 3D plane.

Trellis Graphics Trellis graphs allow you to view relationships between different variables in your data set through conditioning. Chapter 3, *Traditional Trellis Graphics*, provides examples of creating Trellis graphics.

Introduction Suppose you have a data set based on multiple variables and you want to see how plots of two variables change with variations in a third “conditioning” variable. By using Trellis graphics, you can view your data in a series of panels where each panel contains a subset of the original data divided into intervals of the conditioning variable.

For example, a data set contains information on the high school graduation rate per year and average income per household per year for all 50 states. You can plot income against graduation for different regions of the U.S., for example, South, North, East, and West, to determine if the relationship varies geographically. In this case, the regions of the U.S. would be the conditioning variable.

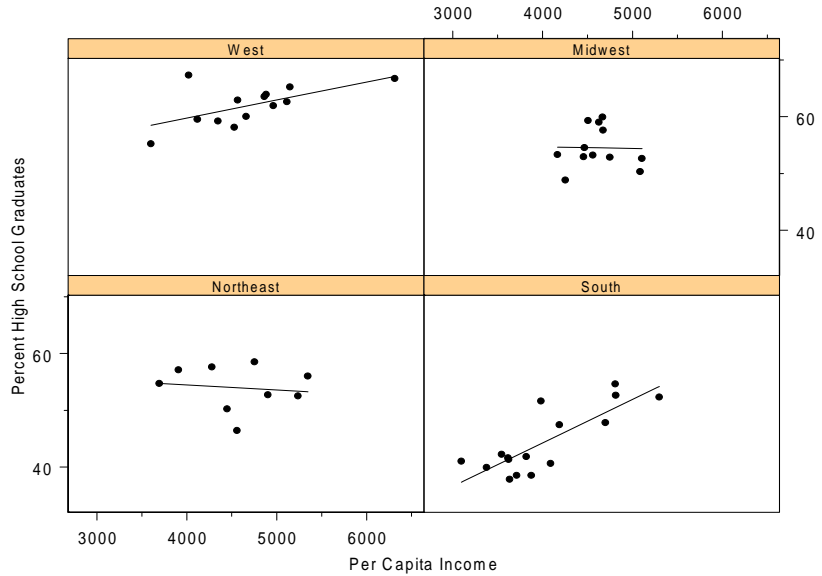


Figure 5.6: A *Trellis* graph.

All graphs can be conditioned using Trellis graphics. The data columns used for the plot and for the conditioning variables must be of equal length. The axis specifications and panel display attributes (for example, fill color) are identical for each panel, although axis ranges may be allowed to vary. The border and fill attributes for the panels can be specified on the **Fill/Border** page of the **Graph** properties dialog.

Creating Trellis Graphs

Creating Trellis graphs using drag-and-drop

1. Create a graph.

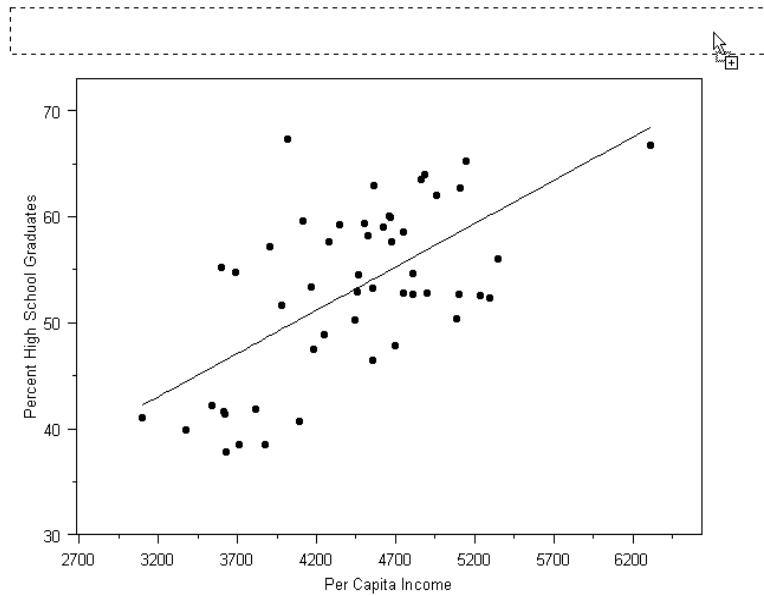


Figure 5.7: *Dragging and dropping conditioning data.*

2. Drag a column of conditioning data onto the graph and drop it in the highlighted rectangle at the top of the graph (the highlighted rectangle is shown in Figure 5.7).
3. If the conditioning data are continuous, you can use the conditioning buttons to change the number of panels.

Creating Trellis graphs using SHIFT-click

1. Create a graph and select the graph area.
2. Select the conditioning data column(s) in the **Data** window or **Object Explorer**.
3. SHIFT-click one of the conditioning buttons on the plot palette.

Plots on Trellis Graphs

Plots on Trellis graphs behave very much the way they do on standard graphs. You can:


1. Double-click or right-click to change the data specifications or any other attributes. When plot specifications change, all of the panels are modified.
2. Change the plot type by selecting the plots and clicking a plot button.
3. Drag new data onto them.
4. Add additional plots.

By default, each plot uses the same conditioning variables specified in the multipanel page of the graph to determine which rows of the data set will be used in each panel. This is appropriate when all of the plots on the graph are using data from the same data set, so that the data columns are all of equal length.

Extracting a Panel From a Trellis Plot


You can extract a single panel from any Trellis graph. This is useful, for example, if each panel contains a lot of detail, and you want to examine one panel at a time very carefully.

Extracting a panel from a Trellis plot

1. Click the **Extract Panel** button  on the **Graph Tools** palette.
2. Click anywhere within the panel that you would like to extract.

Conditioning for the graph is turned off, and the **Subset Rows with** expression for the plot is set to the conditioning expression for that panel. You can also extract a panel by choosing **Format ► Extract Panel ► Redraw Graph** from the main menu. To have the panel placed in a separate **Graph Sheet**, choose **Format ► Extract Panel ► New Graph Sheet** from the main menu.

Restoring the Trellis plot

- Click the **Show All Panels** button  on the **Graph Tools** palette.

or

- From the main menu, choose **Format ► Show All Panels**.

FORMATTING A GRAPH

Spotfire S+ generates all plots using reasonable, data-driven defaults. For example, the x -axis and y -axis lengths are determined by the minima and maxima of the data. The axis labels are derived from the descriptions or column names of the data (if they exist). The colors, line types, plot symbols, and other aspects all have reasonable defaults that work with a wide variety of different data. Once a plot has been generated, Spotfire S+ allows you to annotate and customize practically every aspect of the graph.

This powerful feature is called “formatting a graph.” For example, Figure 5.8 is a default linear fit plot.

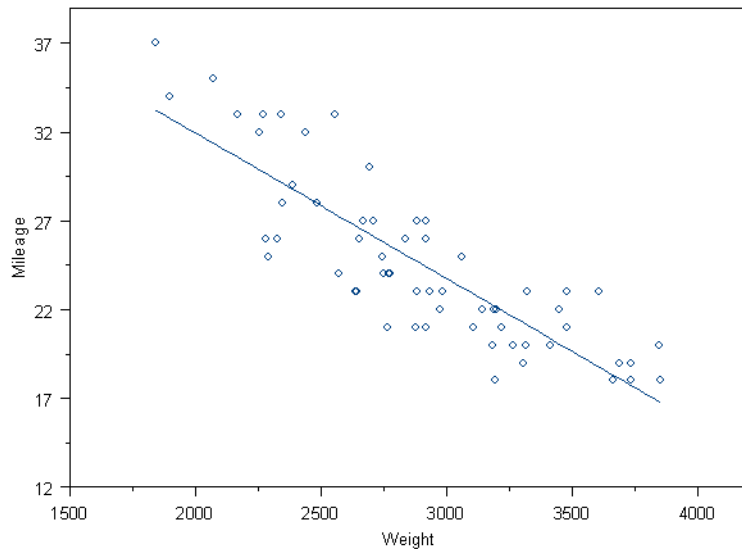


Figure 5.8: *An unformatted linear fit plot.*

After formatting the plot by adding a title, changing the axis tick marks, and adding some annotations, the plot is transformed into Figure 5.9. In this section, we will discuss formatting features and show how Figure 5.8 was transformed into Figure 5.9. For more detail, see the online help.

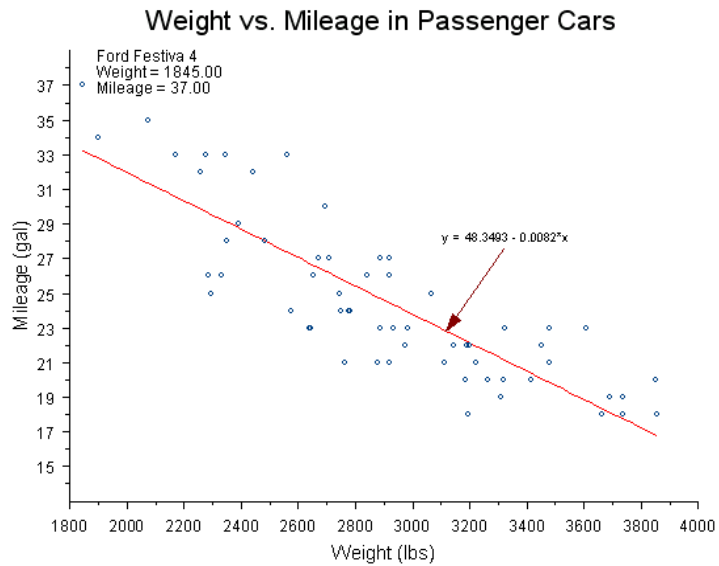


Figure 5.9: *The plot from Figure 5.8 after it has been formatted.*

**Object-Oriented
Nature of Graphs**

All graphics are built from objects that can be individually edited. For example, a line/scatter graph might contain the plot object (the object that plots the data), two axis objects (the objects that define and draw the axes), two axis label objects (the objects that display the axis labels), the title object (the object that displays the graph title), the 2D graph object that defines the general layout, and multipanel options for two-dimensional plots. Each of these objects has its own properties dialog and can be edited. This allows you to customize practically every aspect of the graph. Since all the components in the graph are objects, it is possible to add additional objects (or delete them). For example, a third axis can be added to any plot by adding an additional axis object to the graph.

Formatting a Graph: An Example


The following steps were used to transform Figure 5.8 into Figure 5.9.

Generating the plot

We begin by generating the plot we will format.

1. From the main menu, choose **Data ► Select Data** to open the **Select Data** dialog. Type **fuel.frame** in the **Name** field of the **Existing Data** group and click **OK**.

The `fuel.frame` data set appears in a **Data** window. The data list 60 cars with their respective weight (in pounds), engine displacement, gas mileage, fuel value, and car type. We want to plot the weight versus the gas mileage of the 60 cars. We are interested in generating a plot that helps us address the question: “Does the weight of a car affect the gas mileage and, if so, how?”

2. Select the two data columns `Weight` and `Mileage`. Since we want `Weight` on the x -axis, we click the `Weight` column header first and then we CTRL-click `Mileage`. The two data columns should now be highlighted.
3. Open the **Plots 2D** palette and click the **Linear Fit** button . Spotfire S+ should now display a plot very similar to Figure 5.8. If the plot looks significantly different, it is possible that the default 2D plot properties have been changed.

Formatting the axis titles

Let’s change the text of the y -axis title. In addition, we want to increase the size of the axis titles to make them stand out a bit more.

4. Click twice on the y -axis title **Mileage**. Change the **@Auto** label to **Mileage (gal)**. Click outside the edit box.
5. With the text selected, use the toolbar to increase the font size to **20**.

The y -axis title is now formatted. Format the x -axis title in a similar manner by changing the title to **Weight (lbs)** and the font size to **20**.

Formatting the axes

We continue by formatting the axis labels. We want to make four formatting changes to each of our axes: adjust the tick mark labels, add minor tick marks, increase the width of the axis lines, and remove the frame on the top and right sides of the plot.

6. Click the x -axis line to select it. A green knob should appear on the center of the axis. If a green triangle appears instead of a knob, you have selected the axis labels. Double-click the x -axis line to open the **X Axis** dialog (or type CTRL-1 after the axis has been selected).
7. To increase the width of the axis, select a **Line Weight** of 1 in the **Axis Display** group of the **Display/Scale** page. Click **Apply** to see the change appear.

Note that you can always click **Apply** to see what a change you've made looks like. If you do not click **Apply**, the changes you have made will not display in the graph until you click **OK** to exit the dialog. We will no longer mention this step.

8. To eliminate the top frame, select **None** for the **Frame** in the **Options** group of the **Display/Scale** page.
9. Click the **Range** tab to display the **Range** page of the dialog. In the **Axis Range** group, type 1800 for the **Axis Minimum** and 4000 for the **Axis Maximum**.
10. We want the tick marks to start at the beginning of the axis so also set the **Tick Range** to have **First Tick** at 1800 and **Last Tick** at 4000.
11. We want a major tick mark every 200 lbs so type 200 as the **Major Tick Placement Interval**. Set the **Interval Type** to **Size**.
12. Click the **Grids/Ticks** tab to display the **Grids/Ticks** page of the dialog. In the **Minor Ticks** group, type 0.05 in the **Length** field and click **OK** to exit the dialog.

The x -axis is now formatted. Format the y -axis in a similar manner. Leave the y -axis range the way it is; that is, skip steps 9 and 10 when formatting the y -axis. Type 2 as the **Major Tick Placement Interval** and set the **Interval Type** to **Size**.

Changing the regression line color

Next, we will change the color of the regression line from the default color to a custom color.

13. Select the linear fit plot by clicking the regression line or any data point. A green knob appears at the bottom of the plot, indicating the data object is selected.

14. Right-click the regression line and select **Line/Symbol** from the shortcut menu. A dialog appears with the **Line/Symbol** tab selected.
15. In the **Line** group, select **Custom** from the **Color** drop-down list. The **Color** selection dialog appears.

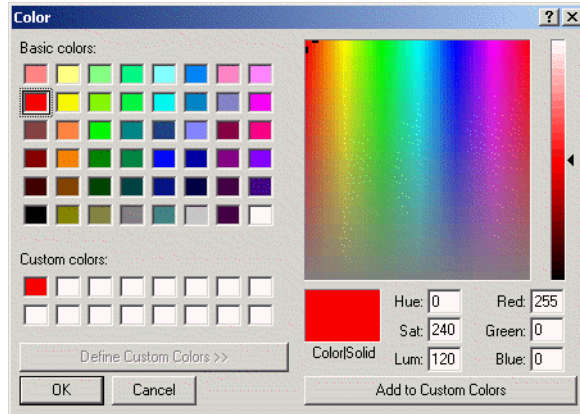


Figure 5.10: Defining a Custom Color

16. You can select the red swatch from the **Basic Colors** group to get basic red, or you can define a custom shade of red. To vary the shade, enter numeric values in the **Red**, **Green**, and **Blue** (RGB) fields. For example, basic red has an RGB value of 255, 0, 0. To define a custom shade of red, experiment by entering other values in the **Green** and **Blue** fields. When you are satisfied with your custom color specification, click **OK**.

Note that on the **Line/Symbol** dialog, the color swatch beside **Custom** in the **Color** drop-down list has changed to match your specification.

17. Click **OK** to apply the custom color to the regression line and dismiss the dialog box.

Adding a title to the graph

To format a title, we need to first insert a title object into the graph.

18. Make sure the **Graph Sheet** is in focus and choose **Insert ► Titles ► Main** from the main menu. An edit box appears. Replace **@Auto** with **Weight vs. Mileage in Passenger Cars**. Do not press RETURN when you are done; instead, click outside the title area. The title will center itself.
19. To increase the font size of the title, click the title to select the title object. Green knobs will surround the title once it has been selected. Use the toolbar to change the size of the font to **28**.

Generally, double-clicking an object will bring up the properties dialog for that object. In the case of a text object (axis, title, or other text object), double-clicking the object causes the object to be edited in-place. Type CTRL-1 to open the properties dialog of a text object instead of double-clicking it as we do with all other objects.


Adding a curve fit equation


We want to add the equation of the regression line to the plot. Spotfire S+ makes this easy.

20. First select the linear fit plot by clicking the regression line or any data point. A green knob should appear at the bottom of the plot, indicating the data object has been selected.
21. From the main menu, select **Insert ► Curve Fit Equation**. The equation of the regression line will appear on the plot. Add **y =** to the front of the equation by clicking twice on the text, pressing HOME, and typing **y =**. Change the font size to **16** using the toolbar. Finally, move the text object by selecting it, left-clicking inside the object (bounded by the green knobs), and dragging it to the desired location.

Annotating the graph

The last step in formatting the graph is to add some annotations to it. We will draw an arrow from the curve fit equation to the line, and we will indicate what car is the most fuel efficient car.

22. Open the **Annotation** palette by clicking the **Annotations** button  on the **Graph** toolbar.

23. First we want to label the most fuel efficient car. Click the **Label Point** button  on the **Annotation** palette; the cursor changes to a large plus sign (see Figure 5.11).

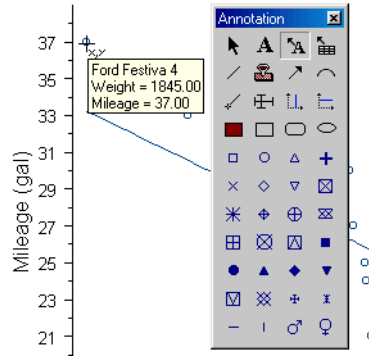




Figure 5.11: Using the **Label Point** tool on the **Annotation** palette.

Move the cursor on top of the top-most y-axis point and click it. The label **Ford Festiva 4** will appear, along with the values of that car's weight and mileage.

24. Click the **Select Tool** button  on the **Annotation** palette, click the label to select it, and use the toolbar to change the label's font size to **16**. If necessary, reposition the label.

25. The final step is to draw the arrow from the curve fit equation to the line. Click the **Arrow Tool** button  on the **Annotation** palette; the cursor changes to a plus sign with an arrow (see Figure 5.12).

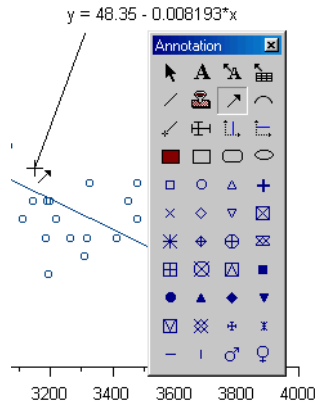


Figure 5.12: Using the **Arrow Tool** on the **Annotation** palette.

Move the cursor to just beneath the minus sign of the curve fit equation. Now drag the mouse from this point (the beginning of the arrow) to just above the regression line. When you release the mouse button, the arrow is drawn.

The formatting is done! The graph on your screen should now resemble Figure 5.9. Later in this chapter, we will continue formatting this graph (see page 288).

Formatting a Graph Sheet

The **Graph Sheet** is the object that defines the plotting environment of the entire page. It is here that you can set the name of the **Graph Sheet**, the plotting orientation (portrait or landscape), the “document units” (default is inches), the size of the **Graph Sheet** (default is 11 by 8.5 inches), the page margins, and the default colors. More advanced users can specify how multiple graphs are plotted on the same **Graph Sheet**.

Formatting the Graph Sheet

1. With the **Graph Sheet** in focus, choose **Format ► Sheet** from the main menu. The **Graph Sheet Properties** dialog appears.

2. Make the desired formatting changes to the **Graph Sheet** and click **OK**.

Saving Graph Sheet Defaults

You can save the settings from your **Graph Sheet** so they will be used as defaults when you create new **Graph Sheets**. This is convenient if you intend to create many **Graph Sheets** with similar properties.

Saving the settings from the active Graph Sheet as the defaults

1. Load a **Graph Sheet** or create a **Graph Sheet** with the desired formatting specifications.
2. Click in a blank area of the **Graph Sheet**, outside any graphs.
3. From the main menu, choose **Options ► Save Window Size/Properties as Default**.

The settings from the current **Graph Sheet** (including window size and position) are saved as the default settings. You can also right-click at the edge of the **Graph Sheet** and select **Save Graph Sheet as default**.

Formatting the Graph

The graph has properties defining the outer bounds of the graph area and the plot area. If you want to resize or move the graph or plot, you can format the graph either interactively or by using the properties dialog.

Formatting the Graph Area

You can select the graph area to change its size or formatting.

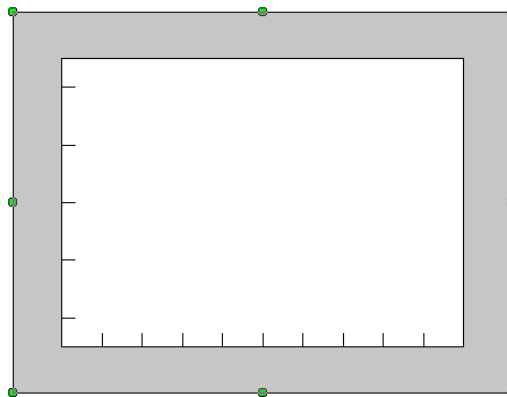


Figure 5.13: *The graph area (gray) is selected.*

Selecting the graph area

- Click anywhere outside the axes but inside the graph boundary. When selected, the graph area has green knobs on all sides and all four corners.

Changing the size of the graph area

- Select the graph area and drag the corner resize knobs to the desired size.

or

- Double-click inside the graph area, but not on another object, to display the **Graph** properties dialog and click the **Position/Size** tab. In the **Graph Size** group, specify the **Width** and **Height** and click **OK**.

or

- Select the graph area, right-click, and select **Position/Size** from the shortcut menu. In the **Graph Size** group, specify the **Width** and **Height** and click **OK**.

Moving the graph area

- Select the graph area and drag it to a new location.

or

- Double-click inside the graph area to display the **Graph** properties dialog and click the **Position/Size** tab. In the **Graph Position** group, specify the **Horizontal** and **Vertical** positions and click **OK**.

Formatting the graph area

- Double-click inside the graph area to display the **Graph** properties dialog and click the **Fill/Border** tab. Make the desired formatting changes to the graph area and click **OK**.

Formatting the Plot Area

You can select the plot area to change its size or formatting. In a 2D graph, the plot area is bounded by the axes.

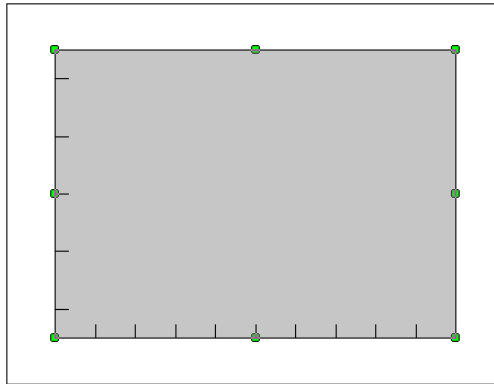


Figure 5.14: *The plot area (gray) is selected.*

Selecting the plot area

- Click anywhere inside the region bounded by the axes. When selected, the plot area has green knobs on all sides and all four corners.

Changing the size of the plot area

- Select the plot area and drag the corner resize knobs to the desired size.

or

- Double-click inside the graph to display the **Graph** properties dialog and click the **Position/Size** tab. In the **Plot Display Size** group, specify the **Width** and **Height** and click **OK**.

or

- Select the plot area, right-click, and select **Position/Size** from the shortcut menu. In the **Plot Display Size** group, specify the **Width** and **Height** and click **OK**.

Moving the plot area

- Select the plot area and drag it to a new location.

or

- Double-click inside the graph to display the **Graph** properties dialog and click the **Position/Size** tab. In the **Plot Origin Position** group, specify **X** and **Y** values and click **OK**.

Formatting the plot area

- Double-click inside the graph to display the **Graph** properties dialog and click the **Fill/Border** tab. Make the desired formatting changes to the plot area border and fill and click **OK**.

Formatting 2D Axes

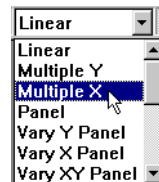
You can customize axes in many ways. You can choose linear, log, or probability axes, in several combinations. Additional axes can be added to your plot, axes can be deleted and moved, and they can be fully customized using the axis properties dialog. Note that Spotfire S+ distinguishes between the axis and the axis label, and each has a separate properties dialog.

Choosing the Axis Type


You can specify the default 2D axis type or specify the axis type as part of formatting an individual axis.





Specifying the default 2D axis type


- From the **Standard** toolbar, select the desired 2D axis type from the dropdown menu.



Interactively Rescaling Axes

You can zoom in on a specified portion of a plot using the **Crop Graph** button . Drag a rectangle around the area of a 2D graph on which you would like to refocus. The x and y axes will be rescaled to show only this area of the graph. This tool can also be accessed by choosing **Format ► Crop Graph** from the main menu.

To see additional regions of the plot, use the pan buttons on the **Graph Tools** palette: , , , .

To restore the full plot, click the **Auto Scale Axes** button  or, equivalently, choose **Format ► Auto Scale Axes** from the main menu.

Selecting an Axis **Selecting an axis**

- Click inside the tick area of the axis. When the axis is selected, it has a square green knob in the center of the axis. If you see a triangular green knob, you have selected the axis labels, not the axis.

Formatting an axis

1. Double-click the axis or select the axis and choose **Format ► Selected Axis** from the main menu. If you want to make the same changes to both axes, select them both using CTRL-click and use either the toolbar or the **Format ► Selected Objects** dialog.
2. In the **Axis** dialog, choose the desired page: **Display/Scale**, **Range**, **Grids/Ticks**, or **Axis Breaks**. Make your changes and click **OK**.

or

- Access pages of the properties dialog by selecting the axis, right-clicking, and choosing one of the axis properties pages from the shortcut menu.

In the **Axis** properties dialog, you can change the display characteristics of your axis. The range and tick mark intervals can be changed, and even the precise look of the major and minor tick marks can be specified. More advanced users can create breaks in the axis.

Moving an axis and offsetting it from the plotting area

- Select the axis. A knob appears in the center of the axis. Drag the knob until the axis is in the desired position.

Adding an additional axis

You can add additional axes to your 2D graph easily. Most easily, you can add a second x or y axis by choosing the **Multiple X** or **Multiple Y** 2D axis type from the **Standard** toolbar.

You can also use drag-and-drop or a menu option to add additional axes:

- Drag one of the extra axes off the **Graph Tools** palette and drop it inside the graph area. An extra axis is added to the selected graph. If an axis already exists in that position, the new axis is automatically offset slightly from the original axis.



Figure 5.15: The extra axis buttons on the **Graph Tools** palette.

or

- Select the graph and choose **Insert ► Axis** from the main menu.

Axes with frames can also be added in the same manner. The frame is not a true axis; it is a mirror of the opposite axis and always has identical scaling.

Formatting 2D Axis Labels

Axis labels are formatted in a way similar to axes. Formatting options include the label type (**Decimal**, **Scientific**, **Percent**, **Currency**, **Months**, **Date**, etc.), the precise location of the tick mark labels, and their font and color. Note that axis titles are not part of the axis specification. They are text objects that can be formatted separately. (See page 280.)

Selecting 2D axis labels

- Click anywhere on the labels to select them. You will see a triangular selection knob. A rectangular green knob means you have selected the axis and not the axis labels.

Formatting 2D axis labels

- Double-click the labels or select the labels and choose **Format ► Selected Axis Labels** from the main menu. In the **Axis Label** dialog, choose the desired page: **Label 1**, **Label 2**, **Minor Labels**, **Position**, or **Font**. Make the desired changes and click **OK**.

Moving 2D axis labels

- Select the axis labels and then drag the labels inside or outside the axis by dragging the triangular selection knob.

or

- Double-click the labels or select the labels and choose **Format ► Selected Axis Labels** from the main menu. Click the **Position** tab, specify values in the **Horizontal** and **Vertical Offset** fields, and click **OK**.


Adding and Formatting Multiline Text

You can add an unlimited amount of multiline text to your graph in the form of text (comments), main titles or subtitles, axis titles, and date and time stamps. The formatting options in the properties dialogs for these different text types are basically the same.

Adding multiline text

- Choose **Insert ► Text** from the main menu. A text edit box will open. (Other types of text, for example, **Titles**, are also available from the **Insert** menu.)

or

- Open the **Annotation** palette and click the **Comment Tool** button ; the cursor changes to the **Comment Tool**. On the **Graph Sheet**, click and drag the cursor and then release the mouse button. Default text the size of the box is added to the graph. Click the selected text to open the edit box.

Type the desired text. Press **ENTER** to create a new line. To end editing and save results, click outside the edit box. Alternatively, you can press **F10** or press **CTRL-ENTER**. To exit without saving, press the **ESC** key.

Editing existing text in-place

- Right-click the text and select **Edit In-place** from the menu.

or

- Click the text to select it, then click again.

Make changes to the text in the edit box. Press ENTER to create a new line. To end editing and save results, click outside the edit box. Alternatively, you can press F10 or press CTRL- ENTER. To exit without saving, press the ESC key.

Moving text

- Select the text. Green selection knobs appear on the outline of the text box. Click *inside* the selected region and drag the box to a new location.

You can also move the text by changing the **X** and **Y** positions on the **Position** page of the properties dialog.

Resizing text

- Select the text. Green selection knobs appear on the outline around the text box. Drag one of the *square* green knobs to increase or decrease the size of the box. The proportions of the text (ratio of height to width) remain constant.

Alternatively, use the toolbar button to change the font size.

Formatting text using the properties dialog

- Select the text and press CTRL-1 or choose **Format ► Selected Comment** from the main menu. The **Comment** dialog appears. Make your desired formatting changes in the dialog and click **OK**.

Formatting text in place

- Open the text edit box and select the text. Choose a **Graph** toolbar option (using the **Font**, **Font Size**, **Bold**, **Underline**, **Italic**, **Superscript**, and **Subscript** buttons) to change the format of the text. You can change the font and point size and choose whether to bold, italicize, or underline the selected text.

or

- Right-click the text to display the shortcut menu. Choose **Superscript** or **Subscript** to format the text, **Font** to open a dialog to edit font type, or **Symbol** to open a dialog to add or edit symbols and Greek characters.

Deleting text

- Select the text. Press the DELETE key. Alternatively, you can select **Clear** from the **Edit** menu or you can right-click and select **Cut**.

Adding Titles and Legends

Inserting a title is different from inserting regular text because a title is positioned automatically. See page 278 for information on editing and formatting titles.

Adding a main title or a subtitle

- From the main menu, choose **Insert ► Titles** and then choose **Main** or **Subtitle**. Spotfire S+ opens an edit box for you to enter text. Type the desired text. Press ENTER to create a new line. To end editing and save results, click outside the edit box. Alternatively, you can press F10 or press CTRL-ENTER. To exit without saving, press the ESC key.

Adding 2D Axis Titles

You can place axis titles on your graph. Axis titles are convenient because they are positioned automatically. See page 278 for information on editing and formatting axis titles.

Adding a 2D axis title

- Select the axis to which you want to add a title. From the main menu, choose **Insert ► Titles ► Axis**. Spotfire S+ opens an edit box for you to enter text. Type the desired text. Press ENTER to create a new line. To end editing and save results, click outside the edit box. Alternatively, you can press F10 or press CTRL-ENTER. To exit without saving, press the ESC key.

Adding 3D Axis Titles

3D axis titles are different from 2D axis titles in that they cannot be moved and sized interactively. The text for 3D axis titles is specified from within the **3D Axes** dialog and cannot be multiline. However, you can add multiline text to 3D graphs in the form of comments and titles. See page 280 for more information on specifying text.


Adding 3D axis titles

- Double-click the axis or select the axis and choose **Format ► Selected 3D Axes** from the main menu. In the **3D Axes** dialog, click the **X Text**, **Y Text**, or **Z Text** tab and make the desired changes for the **Text**, **Font**, **Size**, and **Color** fields. Click **OK**.
- or
- From the main menu, choose **Insert ► Titles ► Axis**. The **3D Axes** dialog appears for editing.

Adding a Date and Time Stamp

A date and time stamp lets you display the date and time along with specified text. See page 278 for information on editing and formatting existing text.


Adding a date stamp

- Click the **Date Stamp Tool** button  on the **Annotation** palette; the cursor changes to the **Date Stamp Tool**. In the **Graph Sheet**, click and drag the **Date Stamp Tool** until you have a text box of the desired size. Release the mouse button. A default date stamp is placed on the graph. The text box will automatically expand if the text starts extending beyond the box boundary.
- or
- From the main menu, choose **Insert ► Annotation ► Date Stamp**. Spotfire S+ opens an edit box for you to enter text. Type the desired text. Press **ENTER** to create a new line. To end editing and save results, click outside the edit box. Alternatively, you can press **F10** or press **CTRL-ENTER**. To exit without saving, press the **ESC** key.

Adding a Legend

A legend is a combination of text and graphics that explains the different plots on the graph. In a multipanel graph, there is only one legend.

Adding a legend

- Click the **Auto Legend** button  on the **Graph** toolbar or choose **Insert ► Legend** from the main menu. If you have more than one graph on the **Graph Sheet**, first select the desired graph before choosing **Legend** from the **Insert** menu.

To remove the legend, click the **Auto Legend** button  again.


Formatting the legend box

- Double-click the legend margin, outside of the legend items, or select the legend and choose **Format ► Selected Legend Item** from the main menu. In the **Legend Item** dialog, you can specify the legend position and formatting for the legend box. Alternatively, you can right-click the legend and select pages of the **Legend Item** dialog from the shortcut menu.

Adding Labels for Points

If you have a 2D scatter plot on your graph, you can automatically display labels (determined by row names) for selected points.


Labeling points

- If you have more than one scatter plot on your **Graph Sheet**, select the scatter plot you want to use. Click the **Label Point** button  on the **Graph Tools** palette. Click a data point in your scatter plot. A label will appear. The label can be moved or edited as any other comment can.
- To replace the label with a label for a different point, click another data point. The first label will be removed and a new label will appear for the newly selected data point.
- To add a label for another point, SHIFT-click another data point. Another label will appear.

Identifying points in a data view

If you have a 2D scatter plot on your graph, you can select rows in the **Data** window by clicking on points in your scatter plot.

- If you have more than one scatter plot on your graph, select the scatter plot you want to use. Open a view on the data used for the x and y columns in the scatter plot. From the main

menu, choose **Window ► Tile Vertical** to show your data and graph side by side. Click the **Select Data Points** button  on the **Graph Tools** palette. Click a data point in your scatter plot. The corresponding row in the **Data** window will become selected.

- To select a different row, click a different point in the scatter plot.
- To add a row to the selection, SHIFT-click another data point.
- To select a group of points, press the left mouse button and drag a rectangle around the points to select.

Adding a Curve Fit Equation

If you have a curve fit plot on your graph, you can automatically display the equation for the line.

Inserting a curve fit equation

- Select the desired curve fit plot. From the main menu, choose **Insert ► Curve Fit Equation**. The equation of the line is displayed on your graph.

Specifying the precision of a curve fit equation

- Right-click the desired curve fit plot and select **Results** from the shortcut menu. The **Curve Fitting Plot** dialog is displayed with the **Results** page in focus. In the **Equation** group, specify the precision to use for your curve fit equation in the **Precision** field (the valid range is 1-15). Click **OK**. From the main menu, choose **Insert ► Curve Fit Equation**. The equation of the line is displayed on your graph with the precision you specified.

Note
Once a curve fit equation has been added to a graph, its precision cannot be changed.

Editing an existing curve fit equation

- Double-click the equation or select the equation and choose **Format ► Selected Comment** from the main menu. For more information on formatting the equation, see page 278.

The curve fit equation option is only available when you have at least one curve fit plot on the graph. If you have multiple curve fit plots, you can select each one and get the equation describing the line automatically.

Adding Lines, Shapes, and Symbols

You can add extra items to your graph, such as text, lines, shapes, and symbols. These drawing objects can be added by using the **Annotation** palette.

Adding an object from the Annotation palette

- Drag-and-drop a drawing object icon from the **Annotation** palette onto the graph.

or

- Click a drawing object button to turn the mouse into a drawing tool. The object to be drawn appears as a small symbol on the lower right side of the mouse pointer. Place this tool on the plot and click and drag to insert a drawing object of a specified size. The object remains selected until you draw another object or click somewhere else on the sheet.

You can resize an object by selecting it and dragging on one of the corner selection knobs or move the object by selecting and dragging in the center of the object. To edit other properties of a drawing object, double-click it to open its properties dialog or right-click it and select the relevant page of the dialog from the shortcut menu.

Modifying Image Colors

For image or contour plots, you can modify the first 16 colors of the image color set that was used when the plot was created.

Converting an image plot to graphic objects

If your plot was created from the command line (e.g., with the `image` or `contour` function), you must first convert the plot to graphic objects before you can modify it in the GUI. For more information, see the note in section *Graphs* on page 250.

Modifying the image colors

- Right-click any object within the plot and select **Contour/Fills** from the shortcut menu. The **Contour Plot** dialog box appears.

Note that **RGB Image Colors** is selected in the **Fill Type** drop-down list.

- Click the **Special Colors** button to display the **Color** selection dialog. The image colors you specified for your plot appear in the 16 color swatches of the **Custom Colors** group. For example, Figure 5.16 shows the 14 custom colors from the `heat.colors` set used by the `image` command to create the plot.

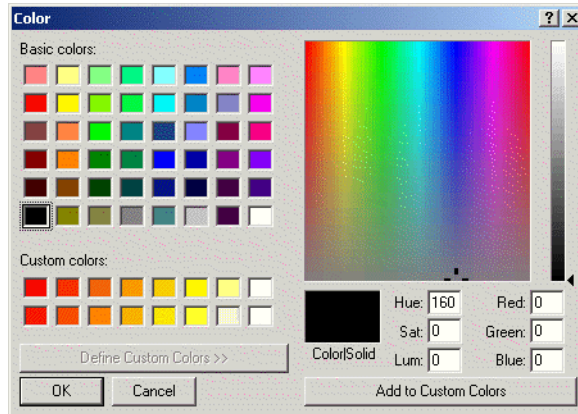


Figure 5.16: *Modifying image colors.*

For information about using `heat.colors` and other color sets provided with Spotfire S+, see the section **Creating Color Sets** on page 13.



- To modify the custom colors, set each swatch to the desired color, then click **OK** to apply your changes and dismiss the **Color** selection dialog.
- On the **Contour Plot** dialog box, click **Apply** to apply your changes to the plot.

WORKING WITH GRAPH OBJECTS

Overlapping Objects

Using the Bring to Front or Send to Back buttons


If objects overlap on a graph, some objects may be completely or partially covered by others. You can change the order of overlapping objects by bringing certain objects to the front or sending others to the back.

- Select the object or objects you want to bring to the front or send to the back. Click the **Bring to Front** button  or the **Send to Back** button  on the **Graph** toolbar, or from the main menu, choose **Format ► Bring to Front** or **Send to Back**. To bring objects forward or backward by single increments (one level at a time), choose the **Bring Forward** or **Send Backward** options from the **Format** menu.

Deleting Objects

Deleting an object from a graph

- Select the object or objects you want to delete. Press the DELETE key or choose **Edit ► Clear** from the main menu.

Once an object is deleted, you can undo the deletion immediately. From the main menu, choose **Edit ► Undo** or click the **Undo** button  on the **Standard** toolbar.

Objects can be removed from the **Graph Sheet** but not permanently deleted by using the **Cut** command. The object is placed on the clipboard so that it can be pasted in another location on the current **Graph Sheet** or in another document.

The Object Explorer and Graph Objects

An alternative way of accessing the objects of a graph is through the **Object Explorer**. All the objects of a graph are stored in the **Object Explorer** under the **Graph Sheet** root node. Expanding that node (clicking on the plus sign) will reveal nodes of all the graphs currently being displayed. You can continue expanding the nodes down the

Graph Sheet tree until the individual graph objects are displayed in the right pane of the **Object Explorer**. Double-clicking on a graph object will display the formatting dialog of that graph object.

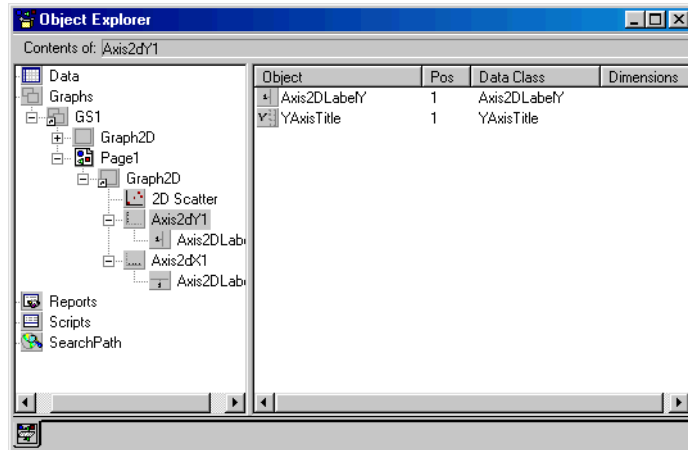


Figure 5.17: *The objects of a graph as seen through the **Object Explorer**.*

The ability to access graph objects is useful when formatting complicated graphs with multiple objects that are overlapping or hidden. There are rare cases where an object can be accessed only through the **Object Explorer**.

PLOT TYPES

So far we have discussed how to create plots. Spotfire S+ is a versatile program, and there are numerous options you can specify in every different plot type Spotfire S+ can plot. For example, in the linear fit plot, the data are plotted and a linear regression line is drawn through the data. It is possible to add confidence limits to that line, change the color of the line, change the color and symbols of the data points, and so on. In this section, we will describe the more commonly used plots and show some of the plot configuration options that let you alter the appearance of the plots. For more detail, see the online help.

Plot Properties Dialog

The first step in changing the plot properties of a plot is to select the plot, just as you would select any other graph object. You select the plot object by clicking on a data point within the plot area. A green knob will appear at the bottom of your plot. If eight green knobs appear surrounding the plot area, then you have selected the plot area instead of the plot itself.

Once you have selected the plot, open the plot properties dialog by doing one of the following:

- Double-click the plot.
- Choose **Format ► Selected Plot** from the main menu.
- Right-click the plot to display its shortcut menu and select a dialog page.

After the plot properties dialog appears, edit the desired properties in the dialog and click **OK**.

Formatting a Graph (Continued)

Earlier in this chapter (see page 266), we went through a step-by-step example on how to format a graph. We will now continue with this process by adding confidence bounds on the regression line and by altering the look of the plotted data. The resulting plot is shown in Figure 5.18.

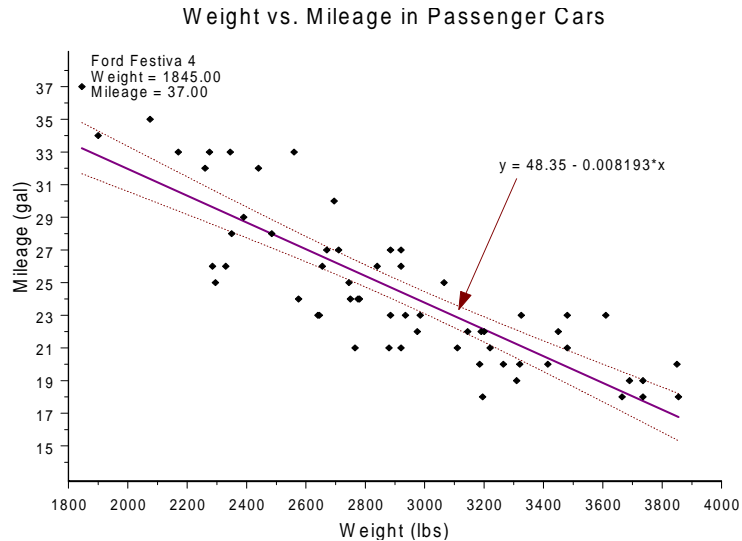


Figure 5.18: Figure 5.9 after we add 95% confidence bounds and change the plot symbols.

To follow this example, you must create the plot shown in Figure 5.8. Follow the steps on page 266 to generate the figure. At a minimum, you must perform steps 1 through 3 to generate the basic plot.

Lines, Symbols, and Colors

1. Select the plot by clicking the regression line or any of the data points. A green knob should appear at the bottom right-hand corner of the plot area, indicating that the plot has been selected. From the main menu, choose **Format ► Selected Curve Fitting Plot** to open the plot properties dialog (or use one of the other methods described earlier in this section).
2. Go to the **Line/Symbol** page and choose **Magenta** for the **Line Color** and **2** for the **Line Weight**. Note how the regression line changes when you click **Apply**.
3. On the same **Line/Symbol** page, select **Diamond, Solid** as the **Symbol Style** and choose **Black** as the **Symbol Color**. Click **Apply**.

Confidence Bounds

Next we add confidence bounds to our plot.

4. Click the **By Conf Bound** tab of the dialog. To add 95% confidence bounds to the regression line, highlight the **Confidence 0.95** line and select the **Line Attributes Style** to

be a series of dots (the last option) and the **Line Attributes Color** to be **Red**. Click **OK** and see the 95% confidence bounds appear on the plot.

Your plot now looks similar to Figure 5.18.

Model Options


The formatting features of the plot properties dialog can be used to explore the characteristics of your data. For example, so far we have fit the simple linear regression line $y = \alpha + \beta x + \varepsilon$. Perhaps you suspect that the regression model $y = \alpha + \beta_1 x + \beta_2 x^2 + \varepsilon$ is more appropriate.

5. To fit this model, open the plot properties dialog and type **2** as the **Polynomial Order** on the **Curve Fitting** page. Note how the curve and confidence bounds change as Spotfire S+ redraws the graph.

Try a few other values (numbers greater than 7 cause problems with this data set). You could also select an exponential or power model to fit to the data instead of the default linear model.

Changing the Plot Type

Extensive formatting is not lost when you change the plot type (see page 253).

6. To change the plot type, make sure the plot is selected (the green knob appears in the bottom right-hand corner). Open the **Plots 2D** palette and click the **Loess** (local regression) button . The graph is redrawn with a loess line drawn through the data.
7. Delete the curve fit equation and arrow (they do not apply to the loess line). Select each of these objects and delete them by choosing **Edit ► Cut** from the main menu.

USING GRAPH STYLES AND CUSTOMIZING COLORS

Graph styles allow you to customize the way your plots will look by default. Two graph styles are available—a **Color Style** and a **Black and White** style. If you use the **Color Style**, plots will be created using different colors. The **Black and White** style distinguishes between plots by using different line and symbol styles but does not vary the color. To specify the graph style to be used for new **Graph Sheets**, choose **Options ► Graph Options** from the main menu and select the desired **Graph Style**. To change a **Graph Sheet** from one style to another, choose **Format ► Apply Style** from the main menu.

You can use the **Options ► Graph Styles** dialogs to customize the two graph styles. On the first page of each dialog, you can specify the color schemes to use, including the scheme for **User Colors** and the background color. The **User Colors** are the 16 extra colors that appear in the dropdown color list when you edit the color of an object. These colors will always be set in a newly created **Graph Sheet** and will override any color specifications in the default **Graph Sheet**. You can edit the background color and color palettes for a specific **Graph Sheet** after it has been created by choosing **Format ► Sheet** from the main menu.

On the first page of the **Graph Styles** dialogs, you can also specify whether or not colors, line styles, patterns, and symbol styles should vary if you put more than one plot on your graph. For example, you might frequently create two line with scatter plots scaled to the same axes. Here you could specify that the plots should be created using different line colors, line styles, and symbol styles but with the same symbol colors.

On the remaining pages of the **Graph Styles** dialogs, you can specify the colors, line styles, patterns, and symbol styles that will be used for plotting the first plot, second plot, etc. if they are varying. If you choose **Plot Default** as the color, Spotfire S+ uses the color settings in the default plot object of the type being created. To modify the plot default, create a plot of the desired type and edit it to attain the desired appearance. Then, on the right-click shortcut menu of the plot, choose **Save [Plot Type] as default**.

There are eight different color schemes available. They can be customized using the **Options ► Color Schemes** dialog. Here, for each color scheme, you can modify the **Background Color**, **User Colors**, and the **Image Colors**. For example, if you edit the **User Colors** of the **Standard** color scheme, all of your newly created **Graph Sheets** will show the revised **User Colors**. The **Image Colors** are a special color palette that can be used for draped surfaces or filled contours. For **Image Colors**, you can specify a number of shades to interpolate between each specified color.

If you like to use different background colors and user colors for different projects, you can edit any or all of the eight color schemes to suit your needs. Once you have set up your color schemes, you can easily switch among them. For example, if you use the **Color Graph Style**, go to **Options ► Graph Styles ► Color**. Set the **User Colors** to **Trellis**, the color scheme traditionally used for Trellis graphics. All new **Graph Sheets** are now created using the **Background Color** and **User Colors** specified in the **Trellis** color scheme. To switch back to using the **Standard** colors by default, just return to **Options ► Graph Styles ► Color** and reset the **User Colors** to **Standard**.

EMBEDDING AND EXTRACTING DATA IN GRAPH SHEETS

You may want to share one of your **Graph Sheets** with a colleague who may want to make further modifications to the graph. To do so, simply embed the data used to create the graph in the **Graph Sheet**. Because the data are embedded, you need only distribute the **Graph Sheet**. (Graphs created by the **Statistics** dialogs have data embedded in them automatically.)

When you embed data in a **Graph Sheet**, only the variables actually displayed in the graph are embedded. So, for example, if you create a plot using four variables from a data set with 20 variables, then embed the data, only four variables are embedded.

You can also extract data from any **Graph Sheet**. This is a convenient way to create new data sets consisting solely of the data used to create a given graph.

Embedding data in a Graph Sheet

1. Create a graph in a **Graph Sheet**.
2. From the main menu, choose **Graph ► Embed Data**. The data required to reproduce the plot are embedded in the **Graph Sheet**.

Once the data are embedded, the **Graph Sheet** is no longer linked to the data set used to create the graph initially. That is, changes to the original data set are not reflected in the **Graph Sheet**. You also cannot use the **Select Data Points** graph tool if data are embedded.

Extracting data from a Graph Sheet

1. Open a **Graph Sheet** displaying data. The plot properties dialog shows **=Internal** as the **Data Set** name for **Graph Sheets** with embedded data, but you can extract data from any **Graph Sheet**.
2. From the main menu, choose **Graph ► Extract Data**. The **Extract Data** dialog appears.
3. Enter a name for the extracted data set and click **OK**.

LINKING AND EMBEDDING OBJECTS

Spotfire S+ supports linking and embedding capabilities so you can use data or objects created in other applications in your **Graph Sheets**. This section describes how to link a Spotfire S+ plot to data from another application and have it retain a connection to the source data.

You should link Spotfire S+ plots to data when:

- the data are likely to change,
- you need the most current version of the data in your Spotfire S+ plot, and
- the source document is available at all times and updating is necessary.

You can also embed Spotfire S+ data or **Graph Sheets** in another application, such as Word or Excel. Embedded objects can be edited using Spotfire S+ from within the other application. This section describes how to embed **Graph Sheets** in another application.

You should embed data or graphics when:

- the embedded information is not likely to change,
- the embedded information does not need to be used in more than one document, and
- the source document is not available for updating if it is linked.

Note

In order to use linking or embedding, the source application must support object linking and embedding (OLE). For example, Excel data can be embedded or linked in a Spotfire S+ **Graph Sheet** because Excel supports OLE.

Data From Another Application

To link data from another application to a Spotfire S+ plot:

1. Select the data in the source application (for example, Microsoft Excel).

2. Copy the data to the clipboard using **Copy** from the **Edit** menu.
3. With your Spotfire S+ plot selected and in focus, choose **Edit ► Paste** from the main menu.

or

- Use the drag-and-drop method by selecting the data from an application and dragging it to the Spotfire S+ plot.

Editing Links

You can control each link in your Spotfire S+ **Graph Sheets**. By default, data are linked to plots with automatic updating. You can change this to manual updating in the **Links** dialog.

Editing links

1. From the main menu, choose **Edit ► Links**.
2. In the **Links** dialog, select the link to edit.
3. Choose **Automatic** or **Manual** linking. Under **Manual** updating, Spotfire S+ only updates the link when you choose the **Update Now** button from the **Links** dialog.

Reconnecting or Changing Links

If you rename or move the source file, you need to reestablish the link. In the **Links** dialog, you need to rename the source file or specify the new location of the source file.

Reestablishing or changing a link

1. From the main menu, choose **Edit ► Links**.
2. Change the name of the linked source file or specify a different file name and click **OK**.

Embedding Spotfire S+ Graphics in Other Applications

Because Spotfire S+ supports object linking and embedding, you can embed Spotfire S+ **Graph Sheets** within other applications, such as PowerPoint and Word.

Embedding a Spotfire S+ Graph Sheet

1. Load the client application, such as Word. Choose **Object** from the **Insert** menu of the client application.

2. Choose the **Create New** button and select Spotfire S+ **Graph Sheet**. Click **OK**. Now you can create and activate the new **Graph Sheet**.
3. When you are finished editing the **Graph Sheet**, click outside the **Graph Sheet** to deactivate it. The embedded **Graph Sheet** is displayed in your document.

Editing the embedded Graph Sheet in place

1. In your client application document, double-click the embedded Spotfire S+ **Graph Sheet**. Alternatively, select the **Graph Sheet**, choose **Object** from the **Edit** menu, and then choose **Edit**.
2. The embedded **Graph Sheet** remains in the client application, but the menus and toolbars change to those used by Spotfire S+. Edit the **Graph Sheet** using the Spotfire S+ menus and toolbars.
3. When finished, click anywhere outside the embedded object to return to the client application's menus and toolbars.

Updating Embedded Graphs

You can update changes to an embedded Spotfire S+ **Graph Sheet** without leaving the current Spotfire S+ session when you have opened an embedded Spotfire S+ **Graph Sheet**.



Updating the embedded Graph Sheet

- Click the **Update** button (this button replaces the **Save** button when editing an embedded **Graph Sheet** in Spotfire S+). The **Update** button updates the embedded graph in the client application document where it is embedded.


or

- Select the **Save Copy As** option on the **File** menu. Save the embedded graph to a new **Graph Sheet** file (*.sgr file) on disk.

PRINTING A GRAPH

To print a **Graph Sheet** in Spotfire S+, use the Windows-standard **Print** button  or the **Print Graph Sheet** option in the **File** menu. You can print a **Graph Sheet** either as a whole or one page at a time. The **Print** button  always prints just the current page, while the **Print** dialog displayed when you choose **Print Graph Sheet** gives you the option of printing all or some of the pages.

Printing a Graph Sheet

- Click the **Print** button  on the **Standard** toolbar. A dialog appears asking you to confirm your print choice.

or

1. From the main menu, choose **File ► Print Graph Sheet**. The **Print** dialog appears.
2. In the **Print** dialog, choose the options that you want. See the online help for a description of the options.
3. Click **OK** to start printing.

EXPORTING A GRAPH TO A FILE

You can export your graphs to files in a wide variety of formats for use in other applications. Note that these files will no longer be linked to Spotfire S+. To share a **Graph Sheet (*.sgr file)** with another application, see Linking and Embedding Objects on page 294.

Exporting a graph to a file

1. Make sure the graph you want to export is showing in the current **Graph Sheet**.
2. From the main menu, choose **File ► Export Graph**. The **Export Graph** dialog is displayed, as shown in Figure 5.19.

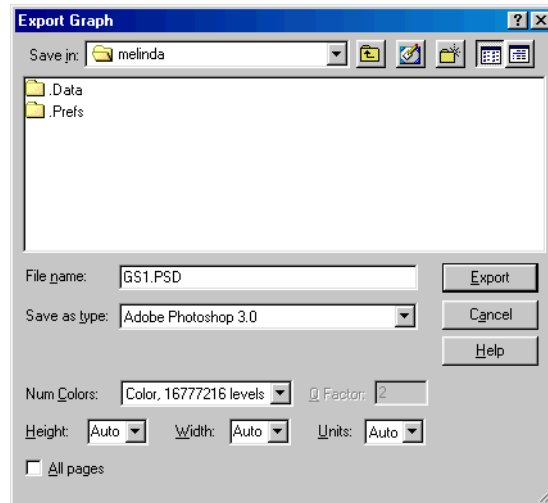


Figure 5.19: The **Export Graph** dialog.

3. Navigate to the desired target folder, if necessary.
4. Select the desired file type in the **Save as type** dropdown list.
5. Type a file name for the exported graph in the **File name** box.
6. Click **Export**.

Exporting Multipage Graph Sheets

If your **Graph Sheet** contains multiple pages and you want to export all the pages simultaneously, do the following:

1. Make sure your multipage **Graph Sheet** is the current document.
2. From the main menu, choose **File ► Export Graph** to open the **Export Graph** dialog.
3. Navigate to the desired target folder, if necessary.
4. Select the desired file type in the **Save as type** dropdown list.
5. Type a “base” file name for the exported graphs in the **File name** box.
6. In the lower left corner of the **Export Graph** dialog, select the **All pages** check box. Selecting this option disables the **File name** field and inserts a # symbol before the last dot in the file name.
7. Click **Export**.

All the **Graph Sheet** pages are exported to separate files, with the # symbol in each file name replaced by its corresponding page number in the **Graph Sheet**.

WINDOWS EDITABLE GRAPHICS COMMANDS

6

Introduction	303
Getting Started	305
Graphics Objects	308
Graph Sheets	308
Graphs	308
Axes	309
Plots	309
Annotations	309
Object Path Names	310
Graphics Commands	313
Plot Types and Plot Classes	313
Viewing Argument Lists and Online Help	315
Specifying Data	317
Display Properties	318
Displaying Dialogs	321
Plot Types	322
The Plots2D and ExtraPlots Palettes	322
The Plots3D Palette	336
Titles and Annotations	345
Titles	345
Legends	345
Other Annotations	346
Locating Positions on a Graph	349
Formatting Axes	350
Formatting Text	352
Modifying the Appearance of Text	353
Superscripts and Subscripts	354
Greek Text	354
Colors	355

Layouts for Multiple Plots	357
Combining Plots on a Graph	357
Multiple Graphs on a Single Page	357
Multiple Graphs on Multiple Pages	358
Conditioned Trellis Graphs	358
Specialized Graphs Using Your Own Computations	359

INTRODUCTION

This chapter introduces the *editable graphics* system that is part of the Spotfire S+ graphical user interface, and it focuses on creating and customizing editable graphics via the point-and-click approach. In this chapter, we show how to create and modify such graphics by calling S-PLUS functions directly. All of the graphics available in the **Plots2D**, **Plots3D**, and **ExtraPlots** palettes can be generated by pointing and clicking, or by typing commands in the **Script** and **Commands** windows. Likewise, editable graphs can be modified by using the appropriate dialogs and the **Graph** toolbar, or by calling functions that make the equivalent modifications.

Note

The graphics produced by the **Statistics** menus and dialogs are *traditional* graphics. See Chapter 2, Traditional Graphics for details.

An editable graph contains numerous *graph objects*, such as axes and annotations. Each field in a graphics dialog corresponds to a property of an object. Similarly, toolbar actions such as changing plot colors or line styles are changes in the values of an object's properties. Each time you create or modify a graph through the Spotfire S+ GUI, the corresponding command is recorded in the **History Log**. This provides an easy way for you to generate programming examples and familiarize yourself with the editable graphics functions. The basic procedure is:

1. Select **Window ► History ► Clear** to clear your **History Log**.
2. Create a graph using the plot palettes and modify it with the dialog and toolbar options.
3. Select **Window ► History ► Display** to view the commands that created your graphic.

By default, Spotfire S+ writes a *condensed History Log*. You can also record a full **History Log** by selecting **Options ► Undo & History** and changing the **History Type** to **Full**.

The main differences between the condensed and full **History Log** are:

- Calls to the `guiPlot` function are recorded in the condensed **History Log** while calls to the `guiCreate` function are recorded in the full **History Log**. We discuss `guiPlot` in the section Getting Started on page 305 and we discuss both functions in the section Graphics Commands on page 313.
- The `guiPlot` calls in the condensed **History Log** include only those arguments that are different from their default values. The `guiCreate` calls in the full **History Log** include all arguments, even if they are not used explicitly to create a particular plot.
- The condensed **History Log** records plotting commands while the full **History Log** also records commands that initialized graph sheets, open palettes, and close dialogs.

Rather than attempt to learn the language of Spotfire S+ editable graphics from scratch, we encourage you to make extensive use of the **History Log** for programming examples and templates. For more information on the **History Log**, see Chapters 9 in the *Spotfire S+ User's Guide for Windows*.

In the section Getting Started on page 305, we provide an overview of the `guiPlot` function and show how it corresponds to particular GUI actions. We then describe the types of objects that constitute an editable graphic in Spotfire S+. We provide examples that show how you can create each type of editable plot programmatically, and then show how you can modify different properties of the plots. Finally, we illustrate how to place multiple plots on a single page, including how to create multipanel Trellis graphics.

In this chapter, we assume that you are familiar with components of the Spotfire S+ graphical user interface such as toolbars, toolbar buttons, palettes, and dialogs.

GETTING STARTED

The `guiPlot` function emulates the action of interactively creating plots by first selecting columns of data and then clicking on a button in a plot palette. The colors, symbol types, and line styles used by `guiPlot` are equivalent to those specified in both the **Options ► Graph Styles** dialogs and the individual graphics dialogs. The arguments to `guiPlot` are:

```
> args(guiPlot)
function(PlotType = "Scatter", NumConditioningVars = 0,
  Multipanel = "Auto", GraphSheet = "", AxisType = "Auto",
  Projection = F, Page = 1, Graph = "New", Rows = "",
  Columns = "", ...)
```

The `PlotType` argument is a character string that matches the name of the plot button as displayed in its tool tip. To see the appropriate string for a plot, hover your mouse over its button in one of the plot palettes until the tool tip appears. Alternatively, use the `guiGetPlotClass` function to see a list of all plot types that `guiPlot` accepts:

```
> guiGetPlotClass()
[1] "Scatter"           "Isolated Points"
[3] "Bubble"           "Color"
[5] "Bubble Color"     "Text as Symbols"
[7] "Line"             "Line Scatter"
[9] "Y Series Lines"   "XY Pair Lines"
[11] "Y Zero Density"   "Horiz Density"
[13] . . .
```

The default value `PlotType="Scatter"` produces a simple scatter plot. For example, the command

```
guiPlot("Scatter", DataSet = "fuel.frame",
  Columns = "Mileage, Weight")
```

emulates the following actions:

1. Highlight the `Mileage` column in the `fuel.frame` data set. CTRL-click to simultaneously highlight the `Weight` column.
2. Click the **Scatter** button in the **Plots2D** palette.

Either approach displays a scatter plot of `Weight` versus `Mileage`.

The `AxisType` argument to `guiPlot` allows you to define different axis types exactly as you do from the **Standard** toolbar. It accepts a string that matches the name of the axis type as it appears in the **Default 2D Axes Type** list. For example, the following call creates a graph with a **Log Y** axis:

```
guiPlot("Scatter", DataSet = "fuel.frame",  
        Columns = "Mileage, Weight", AxisType = "Log Y")
```

This command is equivalent to specifying **Log Y** axes in the **Standard** toolbar before clicking the **Scatter** button in the **Plots2D** palette.

Similarly, the following command creates a graph with two overlaid plots: one showing `Weight` versus `Mileage` and the other showing `Disp.` versus `Mileage`. The `AxisType` argument is set to `"Multiple Y"` so that the y axis for the second plot appears along the right side of the graph sheet, while the y axis for the first plot appears on the left.

```
guiPlot("Scatter", DataSet = "fuel.frame",  
        Columns = "Mileage, Weight, Disp.",  
        AxisType = "Multiple Y")
```

The following call places the plots in two separate panels that have the same x axis scaling but different y axis scaling:

```
guiPlot("Scatter", DataSet = "fuel.frame",  
        Columns = "Mileage, Weight, Disp.",  
        AxisType = "Vary Y Panel")
```

The `NumConditioningVars` argument allows you to create Trellis conditioning plots using `guiPlot`. For example, the command

```
guiPlot("Scatter", DataSet = "fuel.frame",  
        Columns = "Mileage, Weight, Type",  
        NumConditioningVars = 1)
```

emulates the following GUI actions:

1. Click the **Set Conditioning Mode** button in the **Standard** toolbar.
2. Highlight the `Mileage` column in the `fuel.frame` data set. CTRL-click to simultaneously highlight the `Weight` and `Type` columns.
3. Click the **Scatter** button in the **Plots2D** palette.

Either approach creates a scatter plot of Weight versus Mileage for each type of car. The last variable specified in the `Columns` argument to `guiPlot` (or highlighted in the **Data** window) is always used as the conditioning variable. We discuss the `NumConditioningVars` argument more in the section *Conditioned Trellis Graphs* on page 358.

Spotfire S+ writes `guiPlot` commands to the condensed **History Log** when you create a graph interactively. If the **History Type** is set to **Full** instead of **Condensed**, Spotfire S+ writes `guiCreate` commands to the **History Log**; we discuss `guiCreate` more in the section *Graphics Commands* on page 313. You can write your own examples using `guiPlot` by creating the desired plot type and then viewing the condensed **History Log**.

GRAPHICS OBJECTS

There are five main types of graphics objects in the editable graphics system: graph sheets, graphs, axes, plots, and annotations. Plots are contained in graphs, and graphs are contained in graph sheets. Most graphics objects cannot exist in isolation. If a graphics object is created in isolation, it generates an appropriate container. For example, when you create a plot, the appropriate graph, axes and graph sheet are automatically configured and displayed.

In general, the simplest way to create plots is with `guiPlot`. You can create all types of graphics objects with the `guiCreate` function. The properties of graphics objects can be modified using the `guiModify` function. In this section, we briefly describe each of the graphics objects; the section Graphics Commands on page 313 discusses `guiPlot`, `guiCreate`, and `guiModify` in more detail.

Graph Sheets

Graph sheets are the highest-level graphics object. They are documents that can be saved, opened, and exported to a wide variety of graphics formats. *Graph sheet properties* determine the orientation and shape of the graph sheet, the units on the axes, the default layout used when new graphs are added, and any custom colors that are available for other objects. Graph sheets typically contain one or more graphs in addition to annotation objects such as text, line segments, arrows, and extra symbols.

Graphs

There are six types of *graphs* in the editable graphics system: 2D, 3D, Matrix, Smith, Polar, and Text. The graph type determines the coordinate system used within the graph:

- A 2D graph can have one or more two-dimensional coordinate systems, each composed of an x and y axis.
- A 3D graph has a single three-dimensional coordinate system defined by a 3D axes object.
- A Matrix graph has a set of two-dimensional coordinate systems drawn in a matrix layout.
- Smith plots are specialized graphs used in microwave engineering that have a single two-dimensional coordinate system.

- A Polar graph has a single polar coordinate system.
- Text graphs display pie charts and have no coordinate system other than the measurements of the graph sheet.

Graph properties determine the size and shape of both the graph area and the plot area. You can fill both areas with colors and include borders around them. All graphs support the Trellis paradigm of displaying multiple panels; the graph properties determine the conditioning data and the layout of the panels. The 3D graphs also have properties that determine the shape and view angle of the 3D workbook.

Axes

The characteristics of the coordinate systems within graphs are set by the properties of *axes objects*. Typically, *axes properties* contain information about the range, tick positions, and display characteristics of an axis, such as line color and line weight. Axes for 2D graphs also have properties that determine scaling and axis breaks. All axes other than those for 2D graphs contain information about tick labels and axis titles; 2D axes contain separate objects for tick labels and axis titles, both of which have their own properties.

Plots

A *plot* contains data specifications and options relating to how the data are displayed. In many cases, a plot determines the type of calculations that Spotfire S+ performs on the data before drawing the plot. A plot is always contained within a graph and is associated with a particular type of coordinate system. For example, a 2D graph can contain any of the following plot types, among others: bar charts, box plots, contour plots, histograms, density plots, dot charts, line plots, and scatter plots. *Plot properties* are components that describe aspects of the plot such as the line style and color.

Annotations

Annotation objects can be placed directly on a graph sheet or included within a graph. If annotations are contained in a graph, Spotfire S+ repositions them as the graph is repositioned on the page. *Annotation properties* control display information such as line color and line style. They also control an annotation's position on the graph or graph sheet; the units that define the position can be either document units as determined by the graph sheet, or axes units as determined by the

graph's coordinate system. Examples of annotations include titles and legends, which we discuss more in the section Titles and Annotations on page 345.

Object Path Names

Every graph object in Spotfire S+ has a unique path name that identifies it. A valid path name has the following components:

- The first component is the name of the graph sheet preceded by \$\$.
- The name of the graph sheet is followed by the graph number or annotation number.
- The name of the graph is followed by the plot number, axis number, or annotation number.
- The name of an annotation can be followed by numbers that correspond to specific components. For example, legends are annotations that can contain legend items, which control the display of individual entries in a legend.
- In 2D graphics, the name of an axis can be followed by numbers that correspond to tick labels or axis titles.
- The name of some plots can be followed by numbers that correspond to particular plot components. For example, confidence intervals are components that are associated with specific curve fit plots.

The components in the path name for a graph object are separated by dollar signs. You can think of the individual components as containers. For example, plots are contained within graphs, and graphs are contained within graph sheets; therefore, the path name \$\$GS1\$1\$1 refers to the first plot in the first graph of the graph sheet named GS1. Likewise, annotations can be contained within graphs, so the path name \$\$GS1\$1\$1 can also refer to the first annotation in the first graph of GS1. Figure 6.1 visually displays this hierarchy of object path names.

If a path name does not include the name of a graph sheet, Spotfire S+ assumes it refers to the current graph sheet instead. The *current* graph sheet is the one that was most recently created, modified, or viewed.

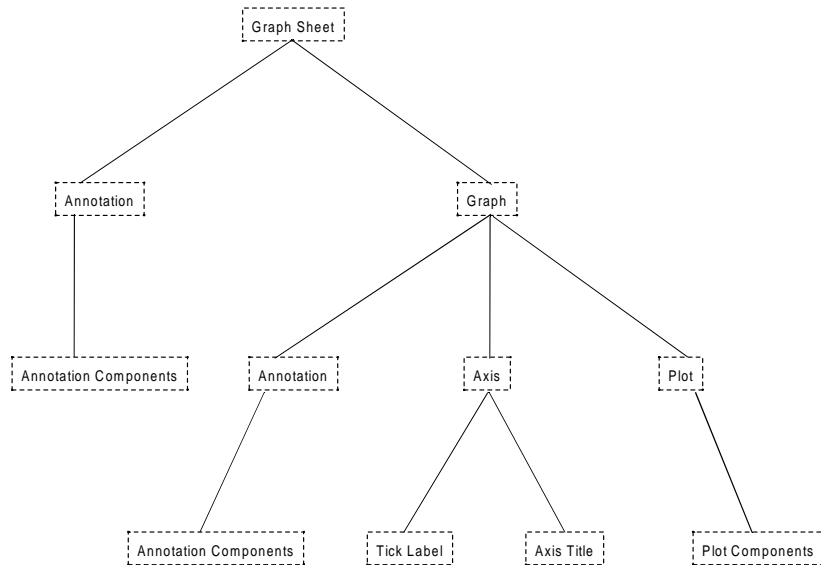


Figure 6.1: *Hierarchy of graph objects in path names. Each node in the tree can be a component of a path name. To construct a full path name for a particular type of graph object, follow a branch in the tree and place dollar signs between the names in the branch.*

You can use the following functions to obtain path names for specific types of graph objects. Most of the functions accept a value for the `GraphSheet` argument, which is a character vector giving the name of the graph sheet. By default, `GraphSheet=""` and the current graph sheet is used.

- `guiGetAxisLabelsName`: Returns the path name of the tick labels for a specified axis. By default, Spotfire S+ returns the path name of the labels for axis 1, which is the first x axis in the first plot on the graph sheet.
- `guiGetAxisName`: Returns the path name of a specified axis. By default, the path name for axis 1 is returned.
- `guiGetAxisTitleName`: Returns the path name of the title for a specified axis. By default, the path name of the title for axis 1 is returned.
- `guiGetGSName`: Returns the path name of the current graph sheet.

- `guiGetGraphName`: Returns the path name of the graph with a specified graph number.
- `guiGetPlotName`: Returns the path name of the plot with the specified graph and plot numbers.

GRAPHICS COMMANDS

This section describes the programming interface to the editable graphics system. The three main functions we discuss are `guiPlot`, `guiCreate`, and `guiModify`. You can use `guiPlot` and `guiCreate` to draw graphics and `guiModify` to change particular properties about your plots. For detailed descriptions of the plot types and their GUI options, see the *User's Guide*.

Throughout this chapter, we emphasize using `guiPlot` over `guiCreate` to generate editable graphics. This is primarily because `guiPlot` is easier to learn for basic plotting purposes. In this section, however, we provide examples using both `guiPlot` and `guiCreate`. The main differences between the two functions are:

- The `guiPlot` function is used exclusively for editable graphics, while `guiCreate` can be used to create other GUI elements such as new **Data** windows and **Object Explorer** pages.
- The `guiPlot` function accepts a *plot type* as an argument while `guiCreate` accepts a *plot class*. We discuss this distinction more in the subsection below.
- Calls to `guiPlot` are recorded in the condensed **History Log** while calls to `guiCreate` are recorded in the full **History Log**.

If you are interested solely in the editable graphics system, we recommend using `guiPlot` to create most of your plots. If you are interested in programmatically customizing the Spotfire S+ graphical user interface, using `guiCreate` to generate graphics may help you become familiar with the syntax of the required function calls.

Plot Types and Plot Classes

Spotfire S+ includes a large number of editable plot types, as evidenced by the collective size of the three plot palettes. Plot types are organized into various *plot classes*, so that the plots in a particular class share a set of common properties. To see a list of all classes for the Spotfire S+ graphical user interface (of which the plot classes are a subset), use the `guiGetClassNames` function.

```
> guiGetClassNames()
[1] "ActiveDocument" "Application" "Arc"
```

```

[4] "Area"           "AreaPanel"       "AreaPlot"
[7] "Arrow"         "attribute"       "Axes3D"
[10] "AxesMatrix"     "AxesPolar"      "Axis2DLabelX"
[13] "Axis2DLabelY"   "Axis2dX"        "Axis2dY"
[16] "AxisPanel"      "AxisPanelLabel" "AxumBoxPlot"
[19] "Bar"            "BarPanel"       "BarPlot"
[22] "Box"           "BoxPanel"       "BoxPlot"
[25] . . .

```

See the section Plot Types on page 322 for comprehensive lists of plots and their corresponding plot classes. Table 6.1 lists the most common classes for the remaining by graph objects (graph sheets, graphs, axes, and annotations).

Table 6.1: Common classes for graph objects. This table does not include plot classes.

Graph Object	GUI Classes
Graph Sheets	GraphSheet, GraphSheetPage, GraphSheetPageItem.
Graphs	Graph2D, Graph3D, GraphMatrix, GraphPolar, GraphSmith, TextGraph.
Axes	Axes3D, AxesMatrix, AxesPolar, Axis3DLabelX, Axis2DLabelY, Axis2dX, Axis2dY.
Titles	MainTitle, Subtitle, XAxisTitle, YAxisTitle.
Legends	Legend, LegendItem, ScaleLegend.
Other Annotations	Arc, Arrow, CommentDate, ConfidenceBound, DateStamp, Ellipse, Radius, ReferenceLine, Slice, Symbol.

You can create variations of basic plot types by modifying the appropriate properties. When creating or modifying a plot, you specify properties by name as arguments to the `guiCreate` and `guiModify` functions. Thus, both `guiCreate` and `guiModify` accept plot classes for their first arguments while `guiPlot` accepts plot types. For example, `Line`, `Scatter`, and `Line Scatter` plots are all members of the plot class `LinePlot`. You can create a scatter plot easily with either `guiPlot` or `guiCreate` as follows:

```
guiPlot("Scatter", DataSet = "fuel.frame",
        Columns = "Weight, Mileage")

guiCreate("LinePlot", DataSet = "fuel.frame",
          xColumn = "Weight", yColumn = "Mileage")
```

Note that `guiPlot` accepts the plot type `Line Scatter` as its first argument while `guiCreate` accepts the plot class `LinePlot`. The `guiCreate` arguments `DataSet`, `xColumn`, and `yColumn` all define properties of a `LinePlot` graphic; they correspond the first three entries on the **Data to Plot** page of the **Line/Scatter Plot** dialog.

To create a line plot with symbols using all of the default values, type:

```
guiPlot("Line Scatter", DataSet = "fuel.frame",
        Columns = "Weight, Mileage")
```

You can generate the same plot with `guiCreate` as follows:

```
guiCreate("LinePlot", DataSet = "fuel.frame",
          xColumn = "Weight", yColumn = "Mileage",
          LineStyle = "Solid")
```

Similarly, you can create a line plot without symbols using either of the following commands:

```
guiPlot("Line", DataSet = "fuel.frame",
        Columns = "Weight, Mileage")

guiCreate("LinePlot", DataSet = "fuel.frame",
          xColumn = "Weight", yColumn = "Mileage",
          LineStyle = "Solid", SymbolStyle = "None")
```

In each of the above examples, Spotfire S+ opens a new graph sheet containing a 2D graph with a set of x and y axes, and then draws the plot within the graph.

Viewing Argument Lists and Online Help

You can obtain online help for `guiPlot` using the `help` function just as you would for any other built-in command. The help files for `guiCreate` and `guiModify` are structured by class name, however. Typing `help(guiCreate)` displays a short, general help file. To see a detailed help page for a `guiCreate` class, you can include the class name in your help call. For example:

```
> help(GCLinePlot)
```

where GC = guiCreate. Note that this syntax does not work for guiModify. To see detailed help for guiModify classes, open the Language Reference online help, click the Index tab, and search for guiModify. Then scroll down to the class name of interest (e.g., guiModify(LinePlot)).

Similarly, you can use the guiPrintClass function to see information about a class. The output from guiPrintClass includes the following:

- A list of arguments for the plot class
- The dialog prompt that corresponds to each argument
- The default value
- Any available options

For example, to see information about the LinePlot class, type:

```
> guiPrintClass("LinePlot")  
  
CLASS:   LinePlot  
ARGUMENTS:  
    Name  
        Prompt:   Name  
        Default:  ""  
    DataSet  
        Prompt:   Data Set  
        Default:  "fuel.frame"  
    xColumn  
        Prompt:   x Columns  
        Default:  ""  
    yColumn  
        Prompt:   y Columns  
        Default:  ""  
    zColumn  
        Prompt:   z Columns  
        Default:  ""  
    . . .
```

The Prompt value gives the name of the field in the **Line/Scatter Plot** dialog that corresponds to each argument. The Default entry gives the default value for the argument, and Option List shows the possible values the argument can assume.

The argument lists for `guiCreate` and `guiModify` are also organized by class name. Instead of using the `args` function to see a list of arguments, use the `guiGetArgumentNames` function. For example, the following command lists the arguments and properties that you can specify for the `LinePlot` class:

```
# The args function does not return a detailed list of
# arguments.
> args(guiCreate)
function(classname, ...)

> args(guiModify)
function(classname, GUI.object, ...)

# The guiGetArgumentNames function returns the arguments
# for a particular plot class.
> guiGetArgumentNames("LinePlot")

[1] "Name"                "DataSet"
[3] "xColumn"             "yColumn"
[5] "zColumn"             "wColumn"
[7] "PlotConditionType"  "ConditionDataSet"
[9] "ConditionColumns"   "PanelToDraw"
[11] "PointLabelsColumn"  "RelativeAxisX"
[13] "RelativeAxisY"      "RelativePlane"
[15] "UseForAspectRatio"  "Hide"
[17] "Crop"                "LineStyle"
[19] "LineColor"          "LineWeight"
[21] . . .
```

You can pass the properties returned by `guiGetArgumentNames` to either `guiCreate` or `guiModify`. Each property corresponds to a field in the dialog for the plot class. The properties returned by the above command all have fields in the **Line/Scatter Plot** dialog.

Specifying Data

You can specify data for plots either *by name* or *by value*. The examples so far in this section illustrate the syntax for specifying data by name. The commands in the examples all refer to data sets and their columns by the associated names, such as "fuel.frame", "Mileage", and "Weight". In this case, the plot is live; it automatically updates when you open it or bring it into focus after the values in the data set have changed. With `guiPlot`, you specify data by name using the `DataSet` and `Columns` arguments, which must be character vectors.

With `guiCreate` and `guiModify`, you specify data by name using the arguments `DataSet`, `xColumn`, `yColumn`, `zColumn` and `wColumn`, all of which accept character vectors.

Alternatively, a plot can store the data internally by value. The expression used to specify the data is evaluated when the plot is created and is not updated thereafter. This is similar to selecting **Graph ► Embed Data** when you wish to embed data in a particular GUI plot. To specify the data values that are used permanently within a plot, use the argument `DataSetValues` in `guiPlot`. With `guiCreate` and `guiModify`, use the arguments `DataSetValues`, `xValues`, `yValues`, `zValues`, and `wValues`. All of these arguments accept S-PLUS expressions such as subscripting statements and data frame names.

For example, to create a scatter plot of `Mileage` versus `Weight` that stores a copy of the data internally in the graph sheet, use one of the following commands:

```
guiPlot("Scatter", DataSetValues =  
  fuel.frame[, c("Mileage","Weight")])  
  
guiCreate("LinePlot",  
  xValues = fuel.frame$Mileage,  
  yValues = fuel.frame$Weight)
```

If you generate plots from within a function, you may want to pass the data by value if you construct the data set in the function as well. Spotfire S+ erases the data upon termination of the function. Therefore, any graphs the function generates by passing the data by name will be empty.

Display Properties

There are a number of display properties commonly used in plots and annotation objects. Table 6.2 lists the properties that determine the appearance of lines and symbols. They correspond to fields in the **Line** and **Symbol** pages of many plot dialogs.

Table 6.2: *Common display properties of plots and annotation objects.*

Property	Description	Settings
LineColor	<p>Color of the lines drawn between data points in the plot.</p> <p>For details about specifying colors, and information about what colors are available, see the section Colors (page 355).</p>	<p>GUI color palette names:</p> <p>"Transparent", "Black", "Blue", "Green", "Cyan", "Red", "Magenta", "Brown", "Lt Gray", "Dark Gray", "Lt Blue", "Lt Green", "Lt Cyan", "Lt Red", "Lt Magenta", "Yellow", "Bright White", "User1", "User2", ..., "User16", "Custom".</p>
LineStyle	<p>Style of the lines drawn between data points in the plot. Accepts a character vector naming the style.</p>	<p>"None", "Solid", "Dots", "Dot Dash", "Short Dash", "Long Dash", "Dot Dot Dash", "Alt Dash", "Med Dash", "Tiny Dash".</p>
LineWeight	<p>Thickness of the lines drawn between data points in the plot. Accepts a numeric value measured in point size.</p>	
SymbolColor	<p>Color of the symbols used to plot the data points.</p> <p>For details about specifying colors, and information about what colors are available, see the section Colors (page 355).</p>	<p>Identical to the settings for LineColor.</p>
FillColor	<p>Color used to fill an object such as a circle or square.</p> <p>For details about specifying colors, and information about what colors are available, see the section Colors (page 355).</p>	<p>Identical to the settings for LineColor.</p>

Table 6.2: Common display properties of plots and annotation objects. (Continued)

Property	Description	Settings
SymbolStyle	Style of the symbols used to plot the data points. Accepts either an integer value representing the style or a character vector naming it.	Integer values: 0,1, 2, ..., 27. Corresponding character values: "None"; "Circle, Solid"; "Circle, Empty"; "Box, Solid"; "Box, Empty"; "Triangle, Up, Solid"; "Triangle, Dn, Solid"; "Triangle, Up, Empty"; "Triangle, Dn, Empty"; "Diamond, Solid"; "Diamond, Empty"; "Plus"; "Cross"; "Ant"; "X"; "-"; " "; "Box X"; "Plus X"; "Diamond X"; "Circle X"; "Box +"; "Diamond +"; "Circle +"; "Tri. Up Down"; "Tri. Up Box"; "Tri. Dn Box"; "Female"; "Male".
SymbolHeight	Size of the symbols used to plot the data points. Accepts a numeric value measured in point size.	

To use the properties listed in the table to change the appearance of your plot, pass them as arguments to either `guiCreate` or `guiModify`. For example, the following commands create a plot of Mileage versus Weight where the points are light red, filled circles and the lines that connect the points are light blue dashes.

```
# Create a basic plot with guiPlot and modify its
# properties with guiModify.
guiPlot("Scatter",
  DataSetValues = fuel.frame[, c("Mileage", "Weight")])
guiModify("LinePlot", Name = guiGetPlotName(),
  LineStyle = "Short Dash",
  LineColor = "Lt Blue",
  LineWeight = "1/2",
  SymbolStyle = "Circle, Solid", SymbolColor = "Lt Red")
```

You can accomplish the same thing using `guiCreate` as follows:

```
# Create a basic line plot with guiCreate and modify its
# properties with guiModify.
guiCreate("LinePlot", xValues = fuel.frame$Mileage,
  yValues = fuel.frame$Weight)
```

```
guiModify("LinePlot", Name = guiGetPlotName(),
  LineStyle = "Short Dash",
  LineColor = "Lt Blue",
  LineWeight = "1/2",
  SymbolStyle = "Circle, Solid", SymbolColor = "Lt Red")
```

In both of the above calls to `guiModify`, the `guiGetPlotName` function returns the path name of the active plot. We discuss path names of GUI objects in the section *Object Path Names* on page 310.

Because you can pass each of the properties in Table 6.2 to `guiCreate` as well as to `guiModify`, you can also draw the plot using a single call to `guiCreate`:

```
guiCreate("LinePlot", xValues = fuel.frame$Mileage,
  yValues = fuel.frame$Weight,
  LineStyle = "Short Dash",
  LineColor = "Lt Blue",
  LineWeight = "1/2",
  SymbolStyle = "Circle, Solid", SymbolColor = "Lt Red")
```

Displaying Dialogs

You can use the `guiDisplayDialog` function to open the property dialog for a particular graph object. For example, the following command displays the dialog for the current plot of class `LinePlot`:

```
guiDisplayDialog("LinePlot", Name = guiGetPlotName())
```

The properties for the plot may be modified using the dialog that appears.

PLOT TYPES

The Spotfire S+ editable graphics system has a wide variety of available plot types. In this section, we present `guiPlot` commands you can use to generate each type of plot. The plots are organized first by palette (**Plots2D**, **ExtraPlots**, and **Plots3D**) and then by plot class. We discuss commands for customizing axes and layout operations in a later section. For additional details on any of the plot types, see the *User's Guide*.

As we mention in the section Getting Started on page 305, you can use the `guiGetPlotClass` function to see a list of all plot types that `guiPlot` accepts. Once you know the name of a particular plot type, you can also use `guiGetPlotClass` to return its class. For example, the `Bubble` plot type belongs to the `LinePlot` class:

```
> guiGetPlotClass("Bubble")
[1] "LinePlot"
```

Knowing both the type and class for a particular plot allows you to use `guiPlot`, `guiCreate`, and `guiModify` interchangeably.

The Plots2D and ExtraPlots Palettes

The **Plots2D** and **ExtraPlots** palettes contain a collection of two-dimensional plots. Table 6.3 shows a quick description of the plot classes and the plots that belong to each of them.

Table 6.3: *The plot types available in the **Plots2D** and **ExtraPlots** palettes. The left column of the table gives the class that each plot type belongs to.*

Plot Class	Description	Available Plot Types
<code>LinePlot</code>	Line and scatter plots.	Scatter, Line, Line Scatter, Isolated Points, Text as Symbols, Bubble, Color, Bubble Color, Vert Step, Horiz Step, XY Pair Scatters, XY Pair Lines, High Density, Horiz Density, Y Zero Density, Robust LTS, Robust MM, Loess, Spline, Supersmooth, Kernel, Y Series Lines, Dot.
<code>LinearCFPlot</code>	Linear curve fit plots.	Linear Fit, Poly Fit, Exp Fit, Power Fit, Ln Fit, Log10 Fit.

Table 6.3: The plot types available in the **Plots2D** and **ExtraPlots** palettes. The left column of the table gives the class that each plot type belongs to.

Plot Class	Description	Available Plot Types
NonlinearCFPlot	Nonlinear curve fit plots.	NLS Fit.
MatrixPlot	Scatterplot matrices.	Scatter Matrix.
BarPlot	Bar plots.	Bar Zero Base, Bar Y Min Base, Grouped Bar, Stacked Bar, Horiz Bar, Grouped Horiz Bar, Stacked Horiz Bar, Bar with Error, Grouped Bar with Error.
HiLowPlot	High-low plots for time series data.	High Low, Candlestick.
BoxPlot	Box plots for a single or grouped variable.	Box, Horizontal Box.
AreaPlot	Area charts.	Area.
QQPlot	One- and two-sample quantile-quantile plots.	QQ Normal, QQ.
PPPlot	One- and two-sample probability plots.	PP Normal, PP.
ParetoPlot	Pareto plots.	Pareto, Horizontal Pareto Plot.
Histogram	Histograms and density curves.	Histogram, Density, Histogram Density.
PiePlot	Pie charts.	Pie.
ErrorBarPlot	Error bar plots.	Error Bar, Horiz Error Bar, Error Bar - Both.
ContourPlot	Contour and level plots.	Contour, Filled Contour, Levels.
VectorPlot	Vector plots.	Vector.

Table 6.3: The plot types available in the **Plots2D** and **ExtraPlots** palettes. The left column of the table gives the class that each plot type belongs to.

Plot Class	Description	Available Plot Types
CommentPlot	Plots in which a third variable can be used to write comments on the graph.	Comment.
SmithPlot	Smith plots.	Smith.
PolarPlot	Polar line and scatter plots.	Polar Line, Polar Scatter.

The LinePlot Class

The `LinePlot` class includes various kinds of line and scatter plots. The *scatter plot* is the fundamental visual technique for viewing and exploring relationships in two-dimensional data. Its extensions include line plots, text plots, bubble plots, step plots, robust linear fits, smooths, and dot plots. The line and scatter plots we illustrate here are the most basic types of plots for displaying data. You can use many of them to plot a single column of data as well as one data column against another.

Scatter plot

```
guiPlot("Scatter", DataSetValues =
  data.frame(util.mktbook, util.earn))
```

Line plot

```
guiPlot("Line", DataSetValues =
  data.frame(util.mktbook, util.earn))
```

Line with scatter plot

```
guiPlot("Line & Scatter", DataSetValues =
  data.frame(util.mktbook, util.earn))
```

Isolated points plot

```
guiPlot("Isolated Points", DataSetValues =
  data.frame(util.mktbook, util.earn))
```

Text as symbols plot

```
guiPlot("Text as Symbols", DataSetValues =
  data.frame(util.mktbook, util.earn, 1:45))
guiModify("LinePlot", Name = guiGetPlotName(),
  SymbolHeight = "0.2")
```

Bubble plot

```
guiPlot("Bubble", DataSetValues =
  data.frame(util.mktbook, util.earn, 1:45))
```

Color plot

```
guiPlot("Color", DataSetValues =
  data.frame(util.mktbook, util.earn, 1:45))
```

Bubble color plot

```
guiPlot("BubbleColor", DataSetValues =
  data.frame(util.mktbook, util.earn, 45:1, 1:45))
```

Vertical step plot

```
guiPlot("Vert Step", DataSetValues =
  data.frame(x = 1:10, y = seq(from=2, to=20, by=2)))
```

Horizontal step plot

```
guiPlot("Horiz Step", DataSetValues =
  data.frame(x = 1:10, y = seq(from=2, to=20, by=2)))
```

XY pairs scatter plot

```
guiPlot("XY Pair Scatters", DataSetValues =
  data.frame(x1 = 1:10, y1 = rnorm(10, mean=1, sd=0.5),
    x2 = 6:15, y2 = rnorm(10, mean=5, sd=0.5)))
```

XY pairs line plot

```
guiPlot("XY Pair Lines", DataSetValues =
  data.frame(x1 = 1:10, y1 = rnorm(10, mean=1, sd=0.5),
    x2 = 6:15, y2 = rnorm(10, mean=5, sd=0.5)))
```

Vertical high density plot

```
guiPlot("High Density", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```

Horizontal high density plot

```
guiPlot("Horiz Density", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```

Y zero high density plot

```
guiPlot("Y Zero Density", DataSetValues =  
  data.frame(x = 1:20, y = runif(20, min=-10, max=10)))
```

Robust least trimmed squares linear fit

```
guiPlot("Robust LTS", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```

Robust MM linear fit

```
guiPlot("Robust MM", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```

Loess smooth

```
guiPlot("Loess", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```

Smoothing spline

```
guiPlot("Spline", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```

Friedman's supersmoother

```
guiPlot("Supersmooth", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```

Kernel smooth

```
guiPlot("Kernel", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```


Y series lines

```
guiPlot("Y Series Lines", DataSet = "hstart",
        Columns = c("Positions", "Data"))
```

Dot plot

```
guiPlot("Dot", DataSetValues =
        data.frame(NumCars = table(fuel.frame$Type),
                  CarType = levels(fuel.frame$Type)))
```

The LinearCFPlot Class

The linear, polynomial, exponential, power, and logarithmic curve fits all have class `LinearCFPlot`. Curve-fitting plots in this class display a regression line with a scatter plot of the associated data points. The curves are computed with an ordinary least-squares algorithm.

Linear fit

```
guiPlot("Linear Fit", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

Polynomial fit

```
guiPlot("Poly Fit", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

Exponential fit

```
guiPlot("Exp Fit", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

Power fit

```
guiPlot("Power Fit", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

Natural logarithmic fit

```
guiPlot("Ln Fit", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

Common logarithmic fit

```
guiPlot("Log10 Fit", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

**The
NonlinearCFPlot
Class**

The `NonlinearCFPlot` class includes a single plot type for fitting nonlinear curves. In addition to the data, this type of plot needs a formula and a vector of initial values for any specified parameters. For this reason, it is usually easier to create the plot with a single call to `guiCreate`, rather than sequential calls to `guiPlot` and `guiModify`.

Nonlinear fit

```
guiCreate("NonlinearCFPlot", DataSet = "Orange",
  xColumn = "age", yColumn = "circumference",
  Model = "circumference ~ A/(1 + exp(-(age-B)/C))",
  Parameters = "A=150, B=600, C=400")
```

**The MatrixPlot
Class**

The `MatrixPlot` class includes a single plot type for displaying *scatterplot matrices*. This type of plot displays an array of pairwise scatter plots illustrating the relationship between any pair of variables in a data set.

Scatterplot matrix

```
guiPlot("Scatter Matrix", DataSet = "fuel.frame",
  Columns = "Mileage, Weight, Type")
```

The BarPlot Class

A wide variety of bar plots are available in the editable graphics system via the `BarPlot` class. A *bar plot* displays a bar for each point in a set of observations, where the height of a bar is determined by the value of the data point. For most ordinary comparisons, we recommend the simplest bar plot with the zero base. For more complicated analysis, you may wish to display grouped bar plots, stacked bar plots, or plots with error bars.

Vertical bar plot with zero base

```
guiPlot("Bar Zero Base", DataSetValues =
  data.frame(as.factor(c("A","B")), c(-20,70)))
```

Vertical bar plot with Y minimum base

```
guiPlot("Bar Y Min Base", DataSetValues =
  data.frame(as.factor(c("A","B")), c(-20,70)))
```

Vertical grouped bar plot

```
guiPlot("Grouped Bar", DataSetValues =
```

```
data.frame(as.factor(c("A","B")), c(20,70), c(30,80)))
guiModify("BarPlot", Name = guiGetPlotName(),
  BarBase = "Zero")
```

Vertical stacked bar plot

```
guiPlot("Stacked Bar", DataSetValues =
  data.frame(as.factor(c("A","B")), c(20,70), c(30,80)))
```

Horizontal bar plot

```
guiPlot("Horiz Bar", DataSetValues =
  data.frame(c(20,70), as.factor(c("A","B"))))
```

Horizontal grouped bar plot

```
guiPlot("Grouped Horiz Bar", DataSetValues =
  data.frame(c(30,80), c(20,70), as.factor(c("A","B"))))
guiModify("BarPlot", Name = guiGetPlotName(),
  BarBase = "Zero")
```

Horizontal stacked bar plot

```
guiPlot("Stacked Horiz Bar", DataSetValues =
  data.frame(c(30,80), c(20,70), as.factor(c("A","B"))))
```

Vertical bar plot with error

```
guiPlot("Bar with Error", DataSetValues =
  data.frame(as.factor(c("A","B")), c(20,70), c(3,6)))
```

Vertical grouped bar plot with error

```
guiPlot("Grouped Bar with Error")
guiModify("BarPlot", Name = guiGetPlotName(),
  xValues = as.factor(c("A","B")),
  yValues = data.frame(c(20,70), c(30,80)),
  zValues = data.frame(c(3,3), c(10,10)))
```

The HiLowPlot Class

The `HiLowPlot` class contains two types of plots: the high-low plot and the candlestick plot. A *high-low plot* typically displays lines indicating the daily, monthly, or yearly extreme values in a time series. These kinds of plots can also include average, opening, and closing values, and are referred to as *high-low-open-close plots* in these cases. Meaningful high-low plots can thus display from three to five

columns of data, and illustrate simultaneously a number of important characteristics about time series data. Because of this, they are most often used to display financial data.

One variation on the high-low plot is the *candlestick plot*. Where typical high-low plots display the opening and closing values of a financial series with lines, candlestick plots use filled rectangles. The color of the rectangle indicates whether the difference is positive or negative. In Spotfire S+, cyan rectangles represent positive differences, when closing values are larger than opening values. Dark blue rectangles indicate negative differences, when opening values are larger than closing values.

High-low-open-close plot

```
dow <- djia[positions(djia) >= timeDate("09/01/87") &
  positions(djia) <= timeDate("11/01/87"), ]
guiPlot("High Low", DataSet = "dow",
  Columns = "Positions, open, close, high, low")
```

Candlestick plot

```
guiPlot("Candlestick", DataSet = "dow",
  Columns = "Positions, open, close, high, low")
```

The BoxPlot Class The `BoxPlot` class contains *box plots* that show the center and spread of a data set as well as any outlying data points. In the editable graphics system, box plots can be created for a single variable or a grouped variable.

Vertical box plot

```
guiPlot("Box", DataSetValues = data.frame(util.earn))
```

Horizontal box plot

```
guiPlot("Horizontal Box", DataSetValues =
  data.frame(util.earn))
```

Vertical grouped box plot

```
guiPlot("Box", DataSet = "fuel.frame",
  Columns = "Type, Mileage")
```

Horizontal grouped box plot

```
guiPlot("Horizontal Box", DataSet = "fuel.frame",
       Columns = "Type, Mileage")
```

The AreaPlot Class

The `AreaPlot` class contains a single plot type that displays area plots. An *area chart* fills the space between adjacent series with color. It is most useful for showing how each series in a data set affects the whole over time.

Area plot

```
guiPlot("Area", DataSetValues =
       data.frame(car.time, car.gals))
```

The QQPlot Class

The `QQPlot` class produces quantile-quantile plots, or *qqplots*, which are extremely powerful tools for determining good approximations to the distributions of data sets. In a one-dimensional qqplot, the ordered data are graphed against quantiles of a known theoretical distribution. If the data points are drawn from the theoretical distribution, the resulting plot is close to the line $y = x$ in shape. The normal distribution is often the distribution used in this type of plot, giving rise to the plot type "QQ Normal". In a two-dimensional qqplot, the ordered values of the variables are plotted against each other. If the variables have the same distribution shape, the points in the qqplot cluster along a straight line.

QQ normal plot

```
# Two data sets compared with the normal distribution.
guiPlot("QQ Normal", DataSetValues =
       data.frame(rnorm(25), runif(25)))
```

QQ plot

```
# Two data sets plotted against each other.
guiPlot("QQ", DataSetValues =
       data.frame(rnorm(25), runif(25)))

# One data set compared with the Chi-square distribution.
guiPlot("QQ", DataSetValues = data.frame(rchisq(20,5)))
guiModify("QQPlot", Name = guiGetPlotName(),
       Function = "Chi-Squared", df1 = "5")
```

The PPPlot Class The `PPPlot` class produces *probability plots*. A one-dimensional probability plot is similar to a `qqplot` except that the ordered data values are plotted against the quantiles of a cumulative probability distribution function. If the hypothesized distribution adequately describes the data, the plotted points fall approximately along a straight line. In a two-dimensional probability plot, the observed cumulative frequencies of both sets of data values are plotted against each other; if the data sets have the same distribution shape, the points in the plot cluster along the line $y = x$.

PP normal plot

```
guiPlot("PP Normal", DataSetValues = data.frame(rnorm(25)))
```

PP plot

```
# Two data sets plotted against each other.
guiPlot("PP", DataSetValues =
  data.frame(rnorm(25), runif(25)))

# One data set compared with the Chi-square distribution.
guiPlot("PP", DataSetValues = data.frame(rchisq(20,5)))
guiModify("PPPlot", Name = guiGetPlotName(),
  Function = "Chi-Squared", df1 = "5")
```

The ParetoPlot Class

The `ParetoPlot` class displays *Pareto charts*, which are essentially specialized histograms. A Pareto chart orders the bars in a histogram from the most frequent to the least frequent, and then overlays a line plot to display the cumulative percentages of the categories. This type of plot is most useful in quality control analysis, where it is generally helpful to focus resources on the problems that occur most frequently. In the examples below, we use the data set `exqcc2` that is located in the `samples\Documents\exqcc2.sdd` file under your Spotfire S+ home directory.

Vertical Pareto plot

```
data.restore(paste(getenv("SHOME"),
  "samples/Documents/exqcc2.sdd", sep = "/"))
guiPlot("Pareto", DataSet = "exqcc2",
  Columns = "NumSample, NumBad")
```

Horizontal Pareto plot

```
guiPlot("Horizontal Pareto Plot", DataSet = "exqcc2",
        Columns = "NumBad, NumSample")
```

The Histogram Class

The Histogram class creates histograms and density plots for one-dimensional data. *Histograms* display the number of data points that fall in each of a specified number of intervals. A *density plot* displays an estimate of the underlying probability density function for a data set and allows you to approximate the probability that your data fall in any interval. A histogram gives an indication of the relative density of the data points along the horizontal axis. For this reason, density plots are often superposed with (scaled) histograms.

Histogram

```
guiPlot("Histogram", DataSetValues = data.frame(util.earn))
```

Density plot

```
guiPlot("Density", DataSetValues = data.frame(util.earn))
```

Histogram with density plot

```
guiPlot("Histogram Density", DataSetValues =
        data.frame(util.earn))
```

The PiePlot Class

The PiePlot class displays *pie charts*, which show the share of individual values in a variable relative to the sum total of all the values. The size of a pie wedge is relative to a sum, and does not directly reflect the magnitude of the data value. Because of this, pie charts are most useful when the emphasis is on an individual item's relation to the whole; in these cases, the sizes of the pie wedges are naturally interpreted as percentages.

Pie chart

```
guiPlot("Pie", DataSetValues =
        data.frame(table(fuel.frame$Type)))
```

The ErrorBarPlot Class

The ErrorBarPlot class includes *error bar plots*, which display a range of error around each plotted data point.

Vertical error bars

```
guiPlot("Error Bar", DataSetValues =  
  data.frame(as.factor(c("A","B")), c(20,70), c(3,6)))
```

Horizontal error bars

```
guiPlot("Horiz Error Bar", DataSetValues =  
  data.frame(c(20,70), as.factor(c("A","B")), c(3,6)))
```

Vertical and horizontal error bars

```
guiPlot("Error Bar - Both", DataSetValues =  
  data.frame(c(20,43), c(20,70), c(3,6), c(5,8)))
```

The ContourPlot Class

The ContourPlot class displays contour plots and level plots. A *contour plot* is a representation of three-dimensional data in a flat, two-dimensional plane. Each contour line represents a height in the z direction from the corresponding three-dimensional surface. A *level plot* is essentially identical to a contour plot, but it has default options that allow you to view a particular surface differently.

Contour plot

```
guiPlot("Contour", DataSet = "exsurf",  
  Columns = "V1, V2, V3")
```

Filled contour plot

```
guiPlot("Filled Contour", DataSet = "exsurf",  
  Columns = "V1, V2, V3")
```

Level plot

```
guiPlot("Levels", DataSet = "exsurf",  
  Columns = "V1, V2, V3")
```

The VectorPlot Class

The VectorPlot class contains the *vector plot* type, which uses arrows to display the direction and velocity of flow at particular positions in a two-dimensional plane. To create a vector plot, specify two columns of data for the positions of the arrows, a third column of data for the angle values (direction), and a fourth column of data for the

magnitude (length). In the example below, we use the data set `exvector` that is located in the `samples\Documents\exvector.sdd` file under your Spotfire S+ home directory.

Vector plot

```
data.restore(paste(getenv("SHOME"),
  "samples/Documents/exvector.sdd", sep = "/"))
guiPlot("Vector", DataSet = "exvector",
  Columns = "x, y, angle, mag")
```

The CommentPlot Class

The `CommentPlot` class contains the *comment plot* type, which displays character labels on a two-dimensional graph. You can use comment plots to display character data, plot combinations of characters as symbols, produce labeled scatter plots, and create tables. To create a comment plot, specify two columns of data for the position of each comment and a third column for the text.

Comment plot

```
guiPlot("Comment", DataSetValues =
  data.frame(x = 1:26, y = rnorm(26), z = LETTERS))
```

The SmithPlot Class

The `SmithPlot` class contains *Smith plots*, which are drawn in polar coordinates. This type of plot is often used in microwave engineering to show impedance characteristics. There are three types of Smith plots: reflection, impedance, and circle. In a reflection plot, the x values are magnitudes in the range $[0,1]$ and the y values are angles in degrees that are measured clockwise from the horizontal. In an impedance plot, the x values are resistance data and the y values are reactance data. In a circle plot, the x values are positive and specify the distance from the center of the Smith plot to the center of the circle you want to draw. The y values are angles that are measured clockwise from the horizontal; the z values are radii and must also be positive.

Smith plots

```
# Reflection plot.
guiPlot("Smith", DataSetValues =
  data.frame(x = seq(from=0, to=1, by=0.1), y = 0:10),
  AxisType="Smith")
guiModify("SmithPlot", Name = guiGetPlotName(),
  AngleUnits = "Radians")
```

```
# Impedance plot.
guiPlot("Smith", DataSetValues =
  data.frame(x = seq(from=0, to=1, by=0.1), y = 0:10),
  AxisType="Smith")
guiModify("SmithPlot", Name = guiGetPlotName(),
  DataType = "Impedance", AngleUnits = "Radians")

# Circle plot.
guiPlot("Smith", DataSetValues =
  data.frame(x = seq(from=0, to=1, by=0.1), y = 0:10,
    z = seq(from=0, to=1, by=0.1)), AxisType="Smith")
guiModify("SmithPlot", Name = guiGetPlotName(),
  DataType = "Circle", AngleUnits = "Radians")
```

The PolarPlot Class

The `PolarPlot` class displays line and scatter plots in polar coordinates. To create a *polar plot*, specify magnitudes for the x values in your data and angles (in radians) for the y values.

Polar line plot

```
guiPlot("Polar Line", DataSetValues = data.frame(
  x = seq(from=0.1, to=2, by=0.1),
  y = seq(from=0.5, to=10, by=0.5)))
```

Polar scatter plot

```
guiPlot("Polar Scatter", DataSetValues = data.frame(
  x = seq(from=0.1, to=2, by=0.1),
  y = seq(from=0.5, to=10, by=0.5)))
```

The Plots3D Palette

The `Plots3D` palette contains a collection of three-dimensional plots. Table 6.4 shows a quick description of the plot classes and the plots that belong to each of them.

The last nine plots in the **Plots3D** palette are composite plots that do not have their own classes. Instead, they are tools that allow you to view plots we've discussed already in new and different ways. The tools fall into two broad categories: *rotated plots* and *conditioned plots*. We discuss each of these categories below.

Table 6.4: The plot types available in the **Plots3D** palette. The left column of the table gives the class that each plot type belongs to.

Plot class	Description	Available Plot Types
Line3DPlot	Line, scatter, drop-line, and regression plots.	3D Scatter, 3D Line, 3D Line Scatter, Drop Line Scatter, 3D Regression, 3D Reg Scatter.
SurfacePlot	Surface and bar plots.	Coarse Surface, Data Grid Surface, Spline Surface, Filled Coarse Surface, Filled Data Grid Surface, Filled Spline Surface, 8 Color Surface, 16 Color Surface, 32 Color Surface, 3D Bar.
ContourPlot	Contour plots. This class contains both 2D and 3D contour plots. See Table 6.3.	3D Contour, 3D Filled Contour.
Grid3D	Projection planes.	This group of plots does not have formal plot types. The plots are listed in the Plots3D palette with the following names: XY Plane Z Min, XZ Plane Y Min, YZ Plane X Min, XY Plane Z Max, XZ Plane Y Max, YZ Plane X Max.
	Rotated plots.	This group of plots has neither a plot class nor a corresponding formal plot type. The plots are listed in the Plots3D palette with the following names: 2 Panel Rotation, 4 Panel Rotation, 6 Panel Rotation.

Table 6.4: The plot types available in the **Plots3D** palette. The left column of the table gives the class that each plot type belongs to.

Plot class	Description	Available Plot Types
	Conditioned plots.	This group of plots has neither a plot class nor a corresponding formal plot type. The plots are listed in the Plots3D palette with the following names: Condition on X, Condition on Y, Condition on Z, No Conditioning, 4 Panel Conditioning, 6 Panel Conditioning.

In the subsections below, we present examples for each of the plot types listed in the table. The data set we use in the examples is created as follows:

```
x <- ozone.xy$x
y <- ozone.xy$y
z <- ozone.median
ozone.df <- data.frame(x,y,z)
```

To familiarize yourself with this data set and the 3D plot types, first create a mesh surface plot:

```
guiPlot("Data Grid Surface", DataSetValues = ozone.df)
```

Next, add the data points as a separate plot to the surface:

```
guiCreate("Line3DPlot", Name = "1$2",
  xValues = x, yValues = y, zValues = z,
  SymbolStyle = "Circle, Solid")
```

The Data Grid Surface is the first plot in the first graph of the graph sheet. We give the plot of data points the name 1\$2 to designate it as the second plot in the first graph. For more details on naming conventions for graph objects, see the section Object Path Names on page 310.

You can use `guiModify` to rotate the axes:

```
guiModify("Graph3D", Name = guiGetGraphName(),
  Rotate3Daxes = T)
```

Note that `Rotate3Daxes` is part of the properties for the graph type `Graph3D` and not the plot type `Line3DPlot`; see the section `Graphics Objects` on page 308 for details.

If you would like to see the surface again without the overlaid data points, use the `guiRemove` function to remove the second plot:

```
guiRemove("Line3DPlot", Name = "1$2")
```

The Line3DPlot Class

The `Line3DPlot` class contains scatter and line plots that display multidimensional data in three-dimensional space. Typically, static 3D scatter and line plots are not effective because the depth cues of single points are insufficient to give strong 3D effects. On some occasions, however, they can be useful for discovering simple relationships between three variables. To improve the depth cues in a 3D scatter plot, you can add drop lines to each of the points; this gives rise to the plot type "Drop Line Scatter". The 3D Regression plot draws a regression plane through the data points.

Scatter plot

```
guiPlot("3D Scatter", DataSetValues = ozone.df)
```

Line plot

```
guiPlot("3D Line", DataSetValues = ozone.df)
```

Line with scatter plot

```
guiPlot("3D Line Scatter", DataSetValues = ozone.df)
```

Drop line scatter plot

```
guiPlot("Drop Line Scatter", DataSetValues = ozone.df)
```

Regression plot

```
guiPlot("3D Regression", DataSetValues = ozone.df)
```

Regression with scatter plot

```
guiPlot("3D Reg Scatter", DataSetValues = ozone.df)
```

The SurfacePlot Class

The `SurfacePlot` class includes different types of *surface plots*, which are approximations to the shapes of three-dimensional data sets. Spline surfaces are smoothed plots of gridded 3D data, and 3D bar plots are gridded surfaces drawn with bars. For two variables, a 3D bar plot produces a binomial histogram that shows the joint distribution of the data. A color surface plot allows you to specify color fills for the bands or grids in your surface plot.

Coarse surface

```
guiPlot("Coarse Surface", DataSetValues = ozone.df)
```

Data grid surface

```
guiPlot("Data Grid Surface", DataSetValues = ozone.df)
```

Spline surface

```
guiPlot("Spline Surface", DataSetValues = ozone.df)
```

Coarse filled surface

```
guiPlot("Filled Coarse Surface", DataSetValues = ozone.df)
```

Data grid filled surface

```
guiPlot("Filled Data Grid Surface",  
        DataSetValues = ozone.df)
```

Filled spline surface

```
guiPlot("Filled Spline Surface", DataSetValues = ozone.df)
```

Eight color draped surface

```
guiPlot("8 Color Surface", DataSetValues = ozone.df)
```

Sixteen color draped surface

```
guiPlot("16 Color Surface", DataSetValues = ozone.df)
```

Thirty-two color draped surface

```
guiPlot("32 Color Surface", DataSetValues = ozone.df)
```

Bar plot

```
guiPlot("3D Bar", DataSetValues = ozone.df)
```

The ContourPlot Class

The 3D contour plots are identical to 2D contour plots, except that the contour lines are drawn in three-dimensional space instead of on a flat plane. For more details, see the section The ContourPlot Class on page 334.

Contour plot

```
guiPlot("3D Contour", DataSetValues = ozone.df)
```

Filled contour plot

```
guiPlot("3D Filled Contour", DataSetValues = ozone.df)
```

The Grid3D Class

The Grid3D class contains a set of two-dimensional planes you can use either on their own or overlaid on other 3D plots. The class is separated into six plots according to which axis a plane intersects and where. For example, the plot created by the XY Plane Z Min button in the **Plots3D** palette intersects the z axis at its minimum.

The plots in the Grid3D class do not have their own plot types. Instead, they are different variations of the Grid3D class, so that you must use `guiCreate` to generate them. In all of the commands below, we overlay planes on a 3D contour plot of the `ozone.df` data.

XY plane

```
# Minimum Z.
guiPlot("3D Contour", DataSetValues = ozone.df)
guiCreate("Grid3D", Name = guiGetPlotName(),
  ProjectionPlane = "XY", Position = "Min")
```

```
# Maximum Z.
guiPlot("3D Contour", DataSetValues = ozone.df)
guiCreate("Grid3D", Name = guiGetPlotName(),
  ProjectionPlane = "XY", Position = "Max")
```

YZ plane

```
# Minimum X.
guiPlot("3D Contour", DataSetValues = ozone.df)
guiCreate("Grid3D", Name = guiGetPlotName(),
```

```
ProjectionPlane = "YZ", Position = "Min")

# Maximum X.
guiPlot("3D Contour", DataSetValues = ozone.df)
guiCreate("Grid3D", Name = guiGetPlotName(),
  ProjectionPlane = "YZ", Position = "Max")
```

XZ plane

```
# Minimum Y.
guiPlot("3D Contour", DataSetValues = ozone.df)
guiCreate("Grid3D", Name = guiGetPlotName(),
  ProjectionPlane = "XZ", Position = "Min")

# Maximum Y.
guiPlot("3D Contour", DataSetValues = ozone.df)
guiCreate("Grid3D", Name = guiGetPlotName(),
  ProjectionPlane = "XZ", Position = "Max")
```

Rotated Plots

The **Plots3D** palette contains buttons that allow you to see 3D plots rotated in either two, four, or six different ways. By rotating a three-dimensional plot, you gain a better understanding of the overall shape of the data. Note that these plots do not have their own class or plot type, but are instead part of a tool that Spotfire S+ provides for you. To use this tool programmatically, define the `Multipanel` argument in your call to `guiPlot` to be one of `"3DRotate2Panel"`, `"3DRotate4Panel"`, or `"3DRotate6Panel"`.

2 panel rotation

```
guiPlot("Data Grid Surface", DataSetValues = ozone.df,
  Multipanel = "3DRotate2Panel")
```

4 panel rotation

```
guiPlot("Data Grid Surface", DataSetValues = ozone.df,
  Multipanel = "3DRotate4Panel")
```

6 panel rotation

```
guiPlot("Data Grid Surface", DataSetValues = ozone.df,
  Multipanel = "3DRotate6Panel")
```


Conditioned Plots

You can condition three-dimensional graphs in the same manner as two-dimensional graphs. The final six buttons in the **Plots3D** palette provide tools that allow you to condition either on a variable in a 3D plot or on a fourth variable that is external to the plot:

- Conditioning on a plot variable causes Spotfire S+ to slice the plot according to the values of that variable. This corresponds to the buttons *Condition on X*, *Condition on Y*, and *Condition on Z* in the **Plots3D** palette. It is available for 3D surface plots only.
- Conditioning on an external variable provides a way to view how a 3D plot varies according to the values in the fourth variable. This corresponds to the buttons *No Conditioning*, *4 Panel Conditioning*, and *6 Panel Conditioning*.

Note that these plots do not have their own class or plot type, but are instead part of a tool that Spotfire S+ provides for you. To use this tool programmatically, set the argument `PanelType="Condition"` in either `guiCreate` or `guiModify`. As we mention in the section *Graphics Objects* on page 308, conditioning parameters are properties of graph objects; thus, `PanelType` is a property of the `Graph3D` class.

Conditioning on a variable in the plot

```
# Condition on X.
guiPlot("Data Grid Surface", DataSet = "exsurf",
  Columns = "V1, V2, V3")
guiModify("Graph3D", Name = guiGetGraphName(),
  PanelType = "Condition", ConditionColumns = "V1")

# Condition on Y.
guiModify("Graph3D", Name = guiGetGraphName(),
  PanelType = "Condition", ConditionColumns = "V2")

# Condition on Z.
guiModify("Graph3D", Name = guiGetGraphName(),
  PanelType = "Condition", ConditionColumns = "V3")
```

Conditioning on a fourth variable

```
# No conditioning.
guiPlot("3D Scatter", DataSet = "galaxy",
```

```
Columns = "east.west, north.south, velocity")

# 4-panel conditioning.
guiPlot("3D Scatter", DataSet = "galaxy",
  Columns = "east.west, north.south, velocity,
    radial.position",
  NumConditioningVars = 1)
guiModify("Graph3D", Name = guiGetGraphName(),
  PanelType = "Condition", NumberofPanels = "4")

# 6-panel conditioning.
guiModify("Graph3D", Name = guiGetGraphName(),
  PanelType = "Condition", NumberofPanels = "6")

# Back to no conditioning.
guiModify("Graph3D", Name = guiGetGraphName(),
  PanelType = "None")
```

TITLES AND ANNOTATIONS

Titles

All graphs can contain titles and subtitles and all 2D axes contain axis titles. To add titles to your 2D editable graphics, specify properties for the "MainTitle", "Subtitle", "XAxisTitle", and "YAxisTitle" GUI classes. For example, the following commands create a basic scatter plot using `guiPlot` and then add all four types of titles using `guiCreate`:

```
guiPlot("Scatter", DataSetValues =
  data.frame(car.miles, car.gals))
guiCreate("XAxisTitle", Name = "1",
  Title = "Gallons per Trip")
guiCreate("YAxisTitle", Name = "1",
  Title = "Miles per Trip")
guiCreate("MainTitle", Name = "1",
  Title = "Mileage Data")
guiCreate("Subtitle", Name = "1",
  Title = "Miles versus Gallons")
```

For 3D graphs, you can use the "MainTitle" and "Subtitle" classes exactly as you do for 2D graphs. Adding axis titles is slightly different, however. This is because 2D axis titles are separate objects with their own properties, while 3D axis titles are themselves properties of 3D axes; we discuss this in the section *Axes* on page 309. Thus, instead of calling `guiCreate` with the "XAxisTitle" and "YAxisTitle" classes, call `guiModify` with the axes class `Axes3D`. For example:

```
xData <- 1:25
guiPlot("3D Line", DataSetValues =
  data.frame(xData, cos(xData), sin(xData)))
guiModify("Axes3D", Name = "1", xTitleText = "x",
  yTitleText = "cos(x)", zTitleText = "sin(x)")
guiCreate("MainTitle", Name = "1", Title = "Spirals")
```

Legends

All graphs can also contain legends. To add a legend to an editable graphic, specify properties for the "Legend" GUI class. This class of graphics objects is equivalent to the legends displayed by the **Auto Legend** button on the **Graph** toolbar. For example, the following

commands create a scatter plot of Weight versus Mileage in the `fuel.frame` data set, vary the plotting symbols according to the `Type` column, and then add a legend.

```
guiPlot("Scatter", DataSet = "fuel.frame",
        Columns = "Weight, Mileage, Type")
guiModify("LinePlot", Name = guiGetPlotName(),
          VarySymbolStyleType = "z Column",
          VarySymbolColorType = "z Column")
guiCreate("Legend", Name = "l$l",
          xPosition = "6.6", yPosition = "6.1")
```

Legends contain legend properties that can be modified individually. To do this, specify properties for the `LegendItem` class of graphics objects. For example, the following command changes the text in the first entry of the legend from above:

```
guiModify("LegendItem", Name = "l$l",
          Text = "Compact Cars")
```

Other Annotations

As we mention in the section Object Path Names on page 310, you can place annotation objects (extra text, lines, symbols, etc.) directly on a graph sheet or within a graph. Because of this, it is necessary to include the graph sheet as part of the `Name` argument when creating these objects. In contrast, commands from previous sections specify names in formats without graph sheet names, similar to `Name="l$l"`. Titles and legends are associated with a particular graph, so the path name defaults to one in the current graph sheet. The annotation objects we discuss in this section, however, are associated with either graph sheets or individual graphs, so the graph sheet must be explicitly included in the object path name.

For example, to center a date stamp at the bottom of a graph sheet, first open the graph sheet and create the appropriate path name for it:

```
> graphsheet()
> gsObjName <- paste("$$", guiGetGSName(), "$1", sep = "")
> gsObjName
[1] "$$GSD2$1"
```

To add a date stamp, specify properties for the `CommentDate` class of GUI objects. Spotfire S+ positions date stamps according to the properties `xPosition` and `yPosition`, using the document units of the

graph sheet as measured from the lower left corner. If a position is set to "Auto", Spotfire S+ centers the date stamp along that axis. For example, the following command centers a date stamp along the width of the graph sheet. For illustration, we use a scatter plot of the built-in data sets `car.gals` and `car.miles`.

```
guiPlot("Scatter", GraphSheet = guiGetGSName(),
  DataSetValues = data.frame(car.gals, car.miles))
guiCreate("CommentDate", Name = gsObjName,
  Title = "My Project",
  xPosition = "Auto", yPosition = 0.1,
  UseDate = T, UseTime = F)
```

The next command places a box with rounded edges outside of the axes in the graph sheet. To do this, we specify properties for the "Box" class of GUI objects.

```
guiCreate("Box", Name = gsObjName,
  OriginX = 0.5, OriginY = 0.5, SizeX = 10, SizeY = 7.5,
  FillColor = "Transparent", RoundCorners = T,
  UseAxesUnits = F)
```

The `OriginX` and `OriginY` arguments position the lower left corner of the box, and `SizeX` and `SizeY` specify its width and length, respectively. The units used are those of the page unless `UseAxesUnits=TRUE`.

Next, create an arrow on the graph. The `xStart`, `yStart`, `xEnd`, and `yEnd` properties define the starting and ending points of the arrow; when `UseAxesUnits=TRUE`, these positions are in axes units. The appropriate path name for the arrow is one level deeper than `gsObjName`, since the arrow is placed directly on the graph:

```
objName <- paste(gsObjName, "$1", sep = "")
guiCreate("Arrow", Name = objName,
  xStart = 15, yStart = 320, xEnd = 20, yEnd = 340,
  UseAxesUnits = T)
```

Similarly, other annotations such as extra symbols and lines can be added:

```
guiCreate("Symbol", Name = objName,
  SymbolStyle = "Box, Empty",
  xPosition = 8, yPosition = 334, SymbolColor = "Red",
  SymbolHeight = 0.25, UseAxesUnits = T)
```

```
guiCreate("Line", Name = objName,  
  LineStyle = "Short Dash",  
  xStart = 5, yStart = 334, xEnd = 25, yEnd = 334,  
  UseAxesUnits=T)
```

The following command adds a horizontal reference line at the mean of the data:

```
guiCreate("ReferenceLine", Name = objName,  
  LineColor = "Black", LineStyle = "Long Dash",  
  Orientation = "Horizontal", Position = mean(car.miles))
```

The next command adds an error bar showing the standard deviation of the data:

```
stddevy <- stdev(car.miles)  
guiCreate("ErrorBar", Name = objName,  
  xPosition = mean(car.gals), yPosition = mean(car.miles),  
  xMin = 0, yMin = stddevy, xMax = 0, yMax = stddevy,  
  UseAxesUnits = T)
```

Other annotation objects such as ellipses, radial lines, and arcs can be used for specialize drawing. The following script creates a new graph sheet and adds such annotations to it:

```
guiCreate("Ellipse", FillColor = "Transparent",  
  xCenter = 5.5, yCenter = 3.5,  
  HorizontalRadius = 2.7, VerticalRadius = 3)  
  
guiCreate("Arc", Name = "1",  
  xCenter = 5.5, yCenter = 2.0,  
  HorizontalRadius = 1.2, VerticalRadius = 0.7,  
  StartAngle = 180, EndAngle = 0)  
  
guiCreate("Radius", Name = "1",  
  xCenter = 5.5, yCenter = 3.3, xStart = 3.0, xEnd = 3.5,  
  Angle = 75)  
  
guiCreate("Radius", Name = "2",  
  xCenter = 5.5, yCenter = 3.3, xStart = 3.0, xEnd = 3.5,  
  Angle = 90)  
  
guiCreate("Radius", Name = "3",  
  xCenter = 5.5, yCenter = 3.3, xStart = 3.0, xEnd = 3.5,  
  Angle = 105)
```

Locating Positions on a Graph

You can use the `guiLocator` function to prompt the user to click on locations in a graph. Among many other things, you can use the chosen locations to interactively place titles, legends, and general annotations. This function accepts as an argument the number of points the user should select. It returns the positions of the chosen points as a list with elements `x` and `y`. By default, `guiLocator` operates on the current graph sheet.

For example, the function `my.rescale` below uses `guiLocator` to allow you to rescale a plot interactively. The `my.rescale` function first creates a line plot with the input data. It then places a comment on the graph prompting the user to click on two points. The `guiLocator` function captures the selected points, which determine the new x axis minimum and maximum. Finally, the comment is removed and the x axis rescaled with the values returned by `guiLocator`.

```
my.rescale <- function(x,y)
{
  guiCreate("LinePlot", xValues = x, yValues = y)
  gsName <- guiGetGSName(),
  commentName <- paste("$$", gsName, "$1", sep = "")
  guiCreate("CommentDate", Name = commentName, Title =
    "Click on two points to determine the \n
    new X axis minimum and maximum",
    xPosition = "Auto", yPosition = 0.12, FontSize = 15,
    UseDate = F, UseTime = F)
  a <- guiLocator(2)
  minXVal <- a$x[1]
  maxXVal <- a$x[2]
  guiRemove("CommentDate", Name = commentName)
  guiModify("Axis2dX", Name = "1$1",
    AxisMin = minXVal, AxisMax = maxXVal)
  invisible()
}
```

To use this function, try the following commands:

```
theta <- seq(from=0, by=pi/10, length=150)
y <- sin(theta)
my.rescale(theta, y)
```

Hint

If you call `guiLocator(-1)`, the current plot redraws.

FORMATTING AXES

You can add axes to a 2D plot by creating objects from the classes `Axis2dX` and `Axis2dY`. The `AxisPlacement` property of these two classes may be set to either "Left/Lower" or "Right/Upper"; this specifies the side of the plot on which to place the axis. The frame for the axis is defined by setting the `DrawFrame` property to "None", "No ticks", "With ticks", or "With labels & ticks". For example, the following commands create a scatter plot and add an x axis with labels and ticks to the top of the plot.

```
guiPlot("Scatter", DataSetValues =
  data.frame(util.mktbook, util.earn))
guiCreate("Axis2dX", Name = "l$l",
  AxisPlacement = "Right/Upper",
  DrawFrame = "With labels & ticks")
```

To customize 3D axes, you must modify the properties of an `Axes3D` object. This is because, unlike 2D axes, 3D axes are single objects that do not contain subclasses. For example, the following commands create a surface plot of the `exsurf` data, change the range of the axes, and modify the axis titles:

```
guiPlot("Data Grid Surface", DataSet = "exsurf",
  Columns = "V1, V2, V3")
guiModify("Axes3D",
  Name = paste(guiGetGraphName(), "$1", sep=""),
  XAxisMin = "DataMin", XAxisMax = "DataMax",
  YAxisMin = "DataMin", YAxisMax = "DataMax",
  ZAxisMin = "DataMin", ZAxisMax = "DataMax",
  XTitleText = "X", YTitleText = "Y", ZTitleText = "Z")
```

The `AxisMin` and `AxisMax` arguments for all three axes accept the character strings "DataMin" and "DataMax" as well as numeric values.

For comparison, the following commands make the same modifications to a two-dimensional graph:

```
XaxesObj <- paste(guiGetGraphName(), "$Axis2dX1", sep="")
guiPlot("Scatter", DataSetValues =
  data.frame(util.mktbook, util.earn))
guiModify("Axis2dX", Name = XaxesObj,
  AxisMin = "DataMin", AxisMax = "DataMax")
```



```
guiModify("XAxisTitle",
  Name = paste(XaxesObj, "$XAxisTitle", sep=""),
  Title = "X")

YaxesObj <- paste(guiGetGraphName(), "$Axis2dY1", sep="")
guiModify("Axis2dY", Name = YaxesObj,
  AxisMin = "DataMin", AxisMax = "DataMax")
guiModify("YAxisTitle",
  Name = paste(YaxesObj, "$YAxisTitle", sep=""),
  Title = "Y")
```

FORMATTING TEXT

You can format the axis labels, titles, and text annotations in your graphs using a set of codes recognized by the Spotfire S+ editable graphics system. Table 6.5 lists the most common text codes and the syntax required for each of them. You can use these codes in `guiCreate` and `guiModify` commands to customize the appearance of text in your editable graphs, as we illustrate in the examples below. The `\\"Symbol\\"` code in the table can be used to include Greek text in graphs as well as other general symbols.

Table 6.5: *Common codes for formatting text in editable graphics.*

Format	Starting Character	Ending Character	Example
Font	\"	\"	\\"Arial\\"
Font size	\p		\p012
Bold	#	#	#HELLO#
Italics	`	`	`Programmer's Guide`
Underline	\u	\u	\uGoodbye\u
Superscript	[]	x[2]
Subscript]	[x]i[
Symbols	\\"Symbol\\"		\\"Symbol\\"abcd
Colors (hex)	"#	"	"#FF0000"
Colors (named)	"=	"	"=olive"
Colors (GUI palette names)	"	"	"Lt Blue"
Extended ASCII character	~		~163

Warning

The backslash is a reserved character in S-PLUS. When you use the codes to change the font size or underlining of text, be sure to include extra backslashes in front of the starting and ending characters: `\\p012` and `\\uGoodbye\\u`.

This also applies when you change the font of or include symbols in your text. Here, however, the two extra backslashes are included at the beginning so that they are not part of the character strings: `\\\\"Arial\\"`, `\\\\"Symbol\\"abcd`.

Like all other features of the editable graphics system, you can use the **History Log** to familiarize yourself with the text codes in the table. The following steps show the idea behind this process:

1. Create a graph from the **Plots2D** palette.
2. Double-click on an axis label so that you can see the `@Auto` string. Type in a different axis label.
3. Highlight the text you wish to modify and right-click to select **Superscript**, **Subscript**, **Font**, or **Symbol** from the context-sensitive menu.
4. When you are finished formatting, click anywhere in the plot to accept the changes.
5. Select **Window ► History ► Display** to view the **History Log**. The formatting you choose appears as text codes in calls to `guiCreate` and `guiModify`.

Modifying the Appearance of Text

The commands below create a scatter plot of two variables in the `fuel.frame` data set, add a `CommentDate`, and modify the text of the annotation. The final `CommentDate` uses Helvetica 20-point font that is bold, italicized, and underlined.

```
guiPlot("Scatter", DataSet = "fuel.frame",
       Columns = "Mileage, Weight")

# Create and position the annotation.
objName <- paste(guiGetGraphName(), "$1", sep="")
guiCreate("CommentDate", Name = objName,
       Title = "The Fuel Data",
       xPosition = "3.9", yPosition = "6.5")
```

```
# Change the appearance of the text.
guiModify("CommentDate", Name = objName,
         Title = "\\\"Helvetica\\\"\\p020#`\\uThe Fuel Data\\u`#")
```

Note the order of the text codes in this command. First, the font name is specified and then the font size. Finally the bold, italics, and underlining surround the text of the title.

Superscripts and Subscripts

The following commands show how you can include superscripts and subscripts in the text of your titles and annotations.

```
x <- seq(from = -2, to = 2, by = 0.1)
y <- x^2
guiPlot("Line", DataSetValues = data.frame(x,y))

guiCreate("MainTitle",
         Name = paste(guiGetGraphName(), "$1", sep=""),
         Title = "x[2] versus x where x[i] = -2, -1.9, ..., 2",
         xPosition = "3.1", yPosition = "7")
```

The title created with this command displays as “ x^2 versus x where $x_i = -2, -1.9, \dots, 2$.”

Greek Text

To include Greek text in the titles and annotations on your graphics, use the `"\Symbol\"` text code. This code precedes the names of the symbols that appear in the **Symbol** table. To access the **Symbol** table, double-click on text in a graph until it is highlighted, right-click and select **Symbol** from the menu. By selecting different symbol types and viewing the **History Log**, you can learn their naming conventions. The names of Greek letters are simply their English counterparts, so that α corresponds to a , β corresponds to b , etc.

For example, the following script plots a parabola, changes the axis titles to the Greek letters α and β , and includes an annotation that displays the equation $\beta = \alpha^2$. In this example, we use the vectors x and y defined in the previous section.

```
guiPlot("Line", DataSetValues = data.frame(x,y))

# Modify the axis titles.
guiModify("XAxisTitle", Name = guiGetAxisTitleName(),
```

```
Title = "\\\"Symbol\"a")
guiModify("YAxisTitle", Name = guiGetAxisTitleName(),
Title = "\\\"Symbol\"b")

# Include the equation of the line as an annotation.
guiCreate("CommentDate", Name = guiGetGraphName(),
Title = "\\\"Symbol\"\\p018b = a[2]",
xPosition = "1.5", yPosition = "5.8")
```

Colors

For general information about available colors, see the section Color Specification (page 3). For details about how colors specified as hexadecimal values or character strings are converted to RGB values, see the section Color Name Resolution (page 9).

Notes

Eight-character hex representations for RGB + Alpha are not supported in GUI commands.

If using Spotfire S+ in use.legacy.graphics(T) mode, see the S-PLUS 7 documentation for an explanation of how to specify colors in GUI command code.

In guiCreate or guiModify commands, you can specify RGB values by using a # prefix to indicate that a hex value follows. For example, to create an Arrow and set its LineColor to red:

```
guiCreate( "Arrow", Name = "GS1$1",
xStart = "4.7978021978022",
yStart = "3.77142857142857",
xEnd = "6.7978021978022",
yEnd = "5.77142857142857",
LineStyle = "Solid",
LineColor = "#FF0000")
```

You can specify a named color by using a = prefix:

```
guiCreate( "Arrow", Name = "GS1$1",
xStart = "4.7978021978022",
yStart = "3.77142857142857",
xEnd = "6.7978021978022",
yEnd = "5.77142857142857",
LineStyle = "Solid",
LineColor = "=blue")
```

If you leave off the = prefix, then the color name refers to the GUI color palette names:

```
guiCreate( "Arrow", Name = "GS1$1",  
xStart = "4.7978021978022",  
yStart = "3.77142857142857",  
xEnd = "6.7978021978022",  
yEnd = "5.77142857142857",  
LineStyle = "Solid",  
LineColor = "Blue")
```

LAYOUTS FOR MULTIPLE PLOTS

Combining Plots on a Graph

You can combine multiple plots within a single graph by defining the `Graph` argument to `guiPlot`. By default, this argument is empty and a new graph is created each time you generate a plot. You can set `Graph` to be the path name of an existing graph, however; this causes Spotfire S+ to place the new plot within the existing graph.

For example, create a line plot of the `car.miles` data with the following command:

```
guiPlot("Line Scatter", DataSet = "car.miles")
```

Add a second plot of a different type and place it in the first graph:

```
guiPlot("Line", DataSet = "car.gals",  
      GraphSheet = guiGetGSName(), Graph = 1)
```

The plot of the `car.miles` data is the first one in the graph. By specifying `Graph=1` in the above command, we place the `car.gals` plot within the first graph.

Note that the ranges of the two data sets are quite different. We can place the plots in separate panels of the graph with the following call to `guiModify`:

```
guiModify("Graph2D", Name = guiGetGraphName(),  
        PanelType = "By Plot")
```

Finally, we can vary the y axis range across panels with the following modification:

```
guiModify("Axis2dY", Name = guiGetAxisName(),  
        VaryAxisRange = T)
```

Multiple Graphs on a Single Page

A graph sheet automatically resizes and repositions any existing graphs on a page when a new graph is added. The layout parameters for positioning the graphs are properties of the graph sheet object. Thus, you can change the arrangement of graphs on a page by specifying the appropriate graph sheet properties. For example, create a new graph sheet containing a line plot with the following command:

```
guiPlot("Line", DataSet = "car.miles")
```

The next command defines the `AutoArrange` property of the graph sheet so that graphs are stacked on top of each other, with one graph in each row:

```
guiModify("GraphSheet", AutoArrange = "One Across")
```

Possible values for the `AutoArrange` property include "None", "One Across", "Two Across", "Three Across", "Four Across", and "Overlaid".

When you add a second graph to the graph sheet, the first graph is moved and resized to fit in the top half of the page while the second graph appears in the lower half:

```
guiPlot("Line", DataSet = "car.gals",  
        GraphSheet = guiGetGSName())
```

Multiple Graphs on Multiple Pages

You can also place multiple graphs on different pages of the same graph sheet. To do this, define the argument `Page="New"` in the call to `guiPlot`. For example, create a new graph sheet containing a line plot with the following command:

```
guiPlot("Line", DataSet = "car.miles")
```

Now add another graph on a second page of the same graph sheet:

```
guiPlot("Line", DataSet = "car.gals",  
        GraphSheet = guiGetGSName(), Page = "New")
```

Conditioned Trellis Graphs

In Trellis graphics, the layout of conditioned plots is specified as part of the graph object. The conditioning variable defined in the `guiPlot` command is used for all plots contained within the graph. For example, create a scatter plot of `Mileage` versus `Weight` conditioned on `Type` in the `fuel.frame` data set:

```
guiPlot("Scatter", DataSet = "fuel.frame",  
        Columns = "Weight, Mileage, Type",  
        NumConditioningVars = 1)
```

The following call to `guiModify` changes the layout so that two plots are placed on each page of the graph sheet:

```
guiModify("Graph2D", Name = "1", NumberOfPages = 3,  
        NumberOfPanelColumns = 2)
```


SPECIALIZED GRAPHS USING YOUR OWN COMPUTATIONS

You can use the `UserFunctionName` property of 2D line plots to create your own customized plot types. For example, the following built-in function creates a plot type that draws crosshairs showing the mean and 95% confidence intervals of x and y data:

```
> crosshairs

function(x, y, z, w, subscripts, panel.num)
{
  # Displays 95% confidence intervals for the means of x and
  # y. x,y,z,w correspond to the data fields in the plot
  # dialog. Currently only numeric columns are supported.
  # z and w can be empty -- NULL will be sent (in this
  # function they are ignored, but in others they might be
  # useful).
  # Subscripts contains the row numbers for the x, y, etc.
  # data. This may be useful for conditioning.
  # panel.num will contain the panel number if
  # conditioning, otherwise it will contain 0.
  meanx <- mean(x)
  meany <- mean(y)
  stdx <- sqrt(var(x))
  stdy <- sqrt(var(y))
  crit.x <- qt(0.975, length(x) - 1)
  crit.y <- qt(0.975, length(y) - 1)
  xdata <- c(meanx - crit.x * stdx, meanx + crit.x * stdx,
            NA, meanx, meanx)
  ydata <- c(meany, meany, NA,
            meany - crit.y * stdy, meany + crit.y * stdy)
  list(x = xdata, y = ydata)
}
```

Notice that the first four arguments of the `crosshairs` function, x , y , z , and w , correspond to the data fields in the plot dialog. If no data are specified for one of these fields, `NULL` is passed into the function. Any user-defined 2D plot type like `crosshairs` must return a list containing the components x and y . Spotfire S+ uses the return vectors as the data to be plotted in the graph.

Note

In this version of `crosshairs`, the arguments `z`, `w`, `subscripts` and `panel.num` are all ignored. However, `subscripts` and `panel.num` may be useful in a version of the function that generates editable Trellis plots. The vector `subscripts` contains the row numbers for the data that are used in the current panel. The argument `panel.num` contains the panel number if conditioning; otherwise it contains 0.

In the code for `crosshairs` above, the first two values in the return vectors `x` and `y` are points that represent a line at the mean of the input data `y`. The boundaries for this line are from the mean of the input `x` minus the 97.5% confidence interval, to the mean of the input `x` plus the 97.5% confidence level. The third element in the return vectors are missing values that are used to break the line. The last two elements in the return values represent a line drawn at the mean of the input `x`, showing the 95% confidence interval for the input `y`.

To create a plot that draws the crosshairs, define the argument `UserFunctionName="crosshairs"` in the call to `guiCreate`. In the command below, we plot the original data points with solid circles so that they are visible in the graph.

```
guiCreate("LinePlot",
  xValues = car.time, yValues = car.miles,
 LineStyle = "Solid", SymbolStyle = "Circle, Solid",
  BreakForMissings = T,
  SmoothingType = "User", UserFunctionName = "crosshairs")
```

Specifying the property `SmoothingType="User"` allows Spotfire S+ to recognize the `UserFunctionName` argument. The `SmoothingType` property is used in any plot that has computations built in; examples include curve fits and smoothed curves as well as user-defined plots.

WORKING WITH GRAPHICS DEVICES

7

Introduction	362
The graphsheet Device	363
Starting a graphsheet Device	363
The motif Device	364
Starting a motif Device	364
The java.graph Device	365
Starting a java.graph Device	365
java.graph Device Settings	365
The pdf.graph Device	374
Creating PDF Files	374
pdf.graph Settings	374
The wmf.graph and emf.graph Devices	383
Creating WMF or EMF Graphics	384
wmf.graph and emf.graph Settings	384
The postscript Device	390
postscript Device Settings	390
Device-Specific Color Specification	397
pdf.graph Colors	397
wmf.graph and emf.graph Colors	401
postscript Colors	403
java.graph Colors	408
Using the S-PLUS 6.2 Colorscheme	411

INTRODUCTION

Spotfire S+ supports several graphic device types. For a complete list, refer to the `Devices` online help file.

This chapter describes the most commonly-used graphics device types:

- `graphsheet` (Windows only)
- `motif` (UNIX/Linux only)
- `java.graph`
- `pdf.graph`
- `wmf.graph`
- `emf.graph`
- `postscript`

Note
The CGM format is not supported in Spotfire S+. To produce CGM graphics, find a package that converts a supported graphic format to CGM.

THE GRAPHSHEET DEVICE

The graphsheet device displays a graphics window that allows you to interactively create, edit, print, and export graphics. It is available on all supported Windows platforms and is the default device type in Spotfire S+ for Windows.

Starting a graphsheet Device

Spotfire S+ sessions begin with no graphics device running. If you have not started a graphics device before executing a plotting command in a Spotfire S+ for Windows session, the graph is automatically sent to the graphsheet device.

To start a graphsheet device manually, call the graphsheet function. The graphsheet arguments and their defaults are:

```
graphsheet(width = NULL, height = NULL, type = "auto",
  win.width = NULL, win.height = NULL, win.left = NULL,
  win.top = NULL, pointsize = 16, orientation = "automatic",
  units.per.inch = 1, format = "", file = "", ncopies = 1,
  color.style = NULL, background.color = NULL,
  print.background = F, color.scheme = NULL,
  image.color.scheme = NULL, color.table = NULL,
  num.image.colors = NULL,
  num.image.shades = if(!is.null(image.color.table)) 10
  else NULL, image.color.table = NULL, pages = NA,
  object.mode = NA, Name = NULL, fonts = character(0),
  suppress.graphics.warnings, params = character(0))
```

For details about these arguments, refer to the graphsheet online help. For information about setting default values, refer to the par online help.

For information about working with graphics in a graphsheet window, see Chapter 5, Editing Graphics in Windows.

THE MOTIF DEVICE

The `motif` device displays a graphics window that allows you to interactively modify the color specifications of your plots and immediately see the result, and also interactively change the plot's print specifications.

The `motif` device conforms to the OSF/Motif graphical user interface standards for the X Window System, Version 11 (X11). It is the default graphics device on all supported UNIX and Linux platforms.

Starting a `motif` Device

Spotfire S+ sessions begin with no graphics device running. If you have not started a graphics device before executing a plotting command in a Spotfire S+ for UNIX/Linux session, the graph is automatically sent to the `motif` device.

To start a `motif` device manually, call the `motif` function. The `motif` arguments and their defaults are:

```
motif(options="", color=T, ignore.sgraphrc=F, ...)
```

For details about these arguments, refer to the `motif` online help. For information about setting default values, refer to the `par` online help.

For information about working with graphics in a `motif` window, see the section Using `motif` Graphics Windows on page 207.

THE JAVA.GRAPH DEVICE

The `java.graph` device is available with Java-enabled versions of Spotfire S+ on all supported operating system platforms. This device displays a graphics window that allows you to interactively modify the color specifications of your plots and immediately see the result, and also interactively change the plot's print specifications.

Starting a java.graph Device

When you start a `java.graph` device (by calling the `java.graph` function) a graphics window is displayed.

The `java.graph` arguments and their defaults are:

```
java.graph(file = "", format = "", width = -1,  
           height = -1, colorscheme = java.colorscheme.default,  
           jpeg.quality = 1., png.color = "color")
```

For details about these arguments, refer to the `java.graph` online help. For information about setting default values for these arguments, refer to the `par` online help.

java.graph Device Settings

You can set the following the `java.graph` device:

- symbols
- color
- font (including point size)
- line types

File formats supported by the `java.graph` device are:

- JPEG (default)
- BMP
- PNG
- PNM
- SPJ
- TIFF

java.graph Symbols

The `pch` argument specifies the number of a plotting symbol to be drawn when plotting points. Table 7.1 lists the available values for the `pch` argument and corresponding plotting symbol.

The symbols for these values are shown in Figure 7.1.

Table 7.1: *java.graph* symbols.

Argument Value	Description
0	square
1	circle
2	triangle
3	cross
4	X
5	diamond
6	inverted triangle
7	square + X
8	X + cross
9	diamond + cross
10	circle + cross
11	triangle + inverted triangle
12	square + cross
13	circle + X
14	square + triangle
15	filled square

Table 7.1: *java.graph* symbols. (Continued)

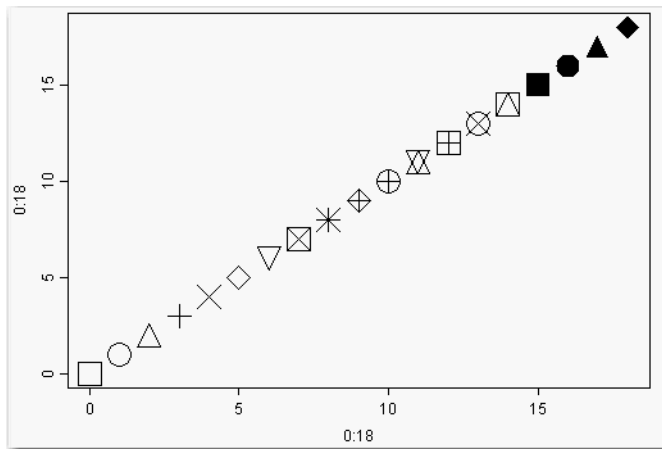
Argument Value	Description
16	filled circle
17	filled triangle
18	filled diamond

These are the same marks described in the `points` online help.

Assigning the values 32 through 126 to the `pch` argument yields the 95 ASCII characters, from space through tilde.

Example

```
java.graph(file="test.jpg")
plot(0:18, 0:18, type="n")
points(0:18, 0:18, pch=0:18, cex=3)
dev.off()
```

**Figure 7.1:** *java.graph* symbols

java.graph Colors By default, Spotfire S+ uses a global color palette and global image color palette to map color indexes to specific colors. Previous Spotfire S+ versions performed color mapping on a device-specific basis. For details about the global palettes, see the section Global Color Palette on page 9.

For Spotfire S+ users who have developed code that makes extensive use of the device-specific palettes, the `use.device.palette` function is available for enabling backward compatibility. For usage details, see the section Backward Compatibility on page 24.

For information about device-specific color controls for the `java.graph` device, see the section `java.graph Colors` on page 408.

java.graph Fonts The following fonts are supported for the `java.graph` device. To use a font, specify its corresponding number for the `font` argument. For example, specify `font=3` for SansSerif bold.

The default font size is 12 points when `cex=1`, but the size is adjusted when the graph is zoomed in and out. It should be exactly 12 points if the graph is 512 pixels wide by 384 pixels high.

Table 7.2: *java.graph fonts.*

Argument value	Font
1	SansSerif plain
2	SansSerif italic
3	SansSerif bold
4	Serif plain
5	Serif italic
6	Serif bold
13	Greek

The following font rules apply:

- The default font is SansSerif plain (`font=1`).

- The font values loop; that is, font=7 is the same as font=1, font=8 is the same as font=2, and so on. The exception to this rule is font=13, which assigns the Greek font.
- Any font number set to less than 1 is the same as font=1. The set of fonts cannot be changed.

Example:

```
java.graph(file="test.jpg")  
plot(1:25, 1:25, type="n")  
text(1:25, 1:25, "qwerty", font=1:25)  
dev.off()
```

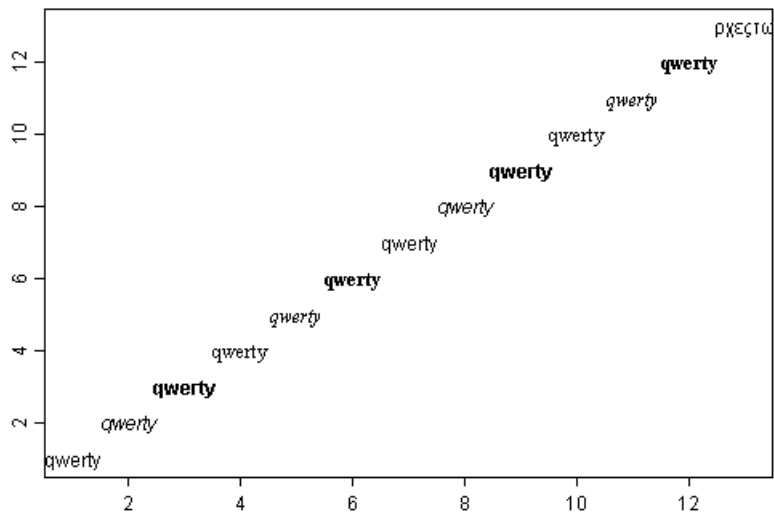


Figure 7.2: *java.graph* fonts.

java.graph Line Types

The line type is controlled by the `lty` argument. The following line types are supported for `java.graph`.

Table 7.3: *java.graph line types.*

Argument value	Line type
1	solid
2	dotted
3	short dashed dotted
4	medium dashed
5	dashed with three dots
6	medium dashed dotted
7	dotted (groups of three)
8	short dashed

Example:

```
java.graph(file="test.jpg")  
plot(1:8, 1:8, type="n")  
for(i in 1:8) lines(c(1,8), c(i,i), lty=i)
```

dev.off()

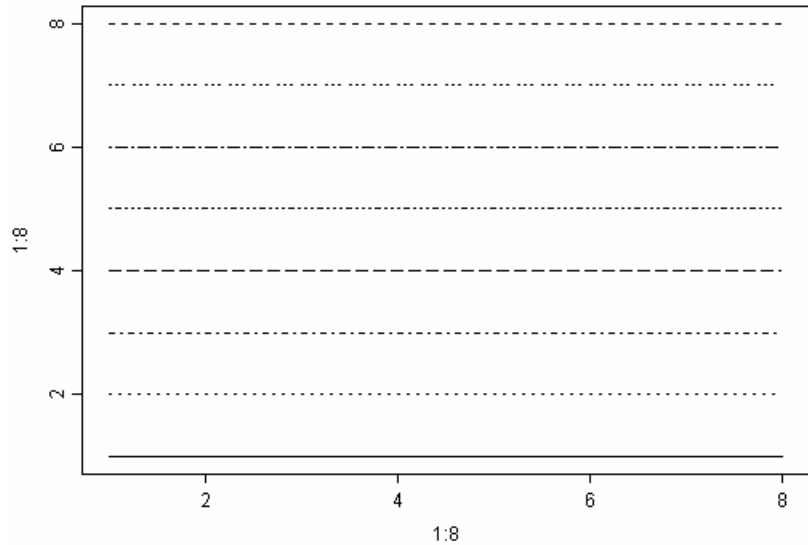


Figure 7.3: *java.graph* line types.

**Resolution (DPI)
Information**

The dpi graph style is dependent on the resolution of the printer or device, not the resolution of the graph produced by Spotfire S+. Spotfire S+ produces 300 dpi graphics.

Resolution is fixed in all Spotfire S+ bitmapped files as a function of page dimensions. To change the resolution, you have two options:

- Change the default resolution of a (TIFF) graphics file from 72 dpi (the default) by creating a WMF graphics file (using `wmf.graph`) or EMF graphics file (`emf.graph`) and converting it to a TIFF in another application that provides the ability to change resolution. Currently, there is not a way to change this in Spotfire S+.
- The `java.graph` function takes the arguments `width` and `height` (in pixels), which you can use to set the size of the bitmap that the graphic is drawn on, before the bitmap is converted to a JPEG or other format. By increasing these values from the defaults, you can create a higher-quality image. Of course, the resulting image files are larger. If just one of `width` or `height` is given, the other is set to keep the normal width/height ratio.

Graphics Formats Details The following guidelines can help you with your graphics formats:

- JPEG, BMP, TIFF, and GIF are bitmapped graphics. They are raster formats more suited for displaying images (pictures) than vector-based graphics. Spotfire S+ displays graphics internally as vector graphics. Vector graphics are redrawn according to scale and do not suffer from pixilation as do bitmaps.
- The 16-bit Windows Metafile (WMF) or the 32-bit Enhanced Metafile (EMF) are the suggested graphics formats because they are vector-based types of files. Graphsheets look sharp inside of Spotfire S+ because vector graphics are used to create images. This means that Spotfire S+ plots are drawn with complete lines. In contrast, JPEG, GIF, TIFF, BMP, and other bitmapped file formats use pixels, and can result in some information loss due to compression. These formats work well with photographs, which usually contain so much information that the loss is not dramatic. However, all of the information in a graphics file is necessary, so Spotfire S+ plots do not tend to export well to bitmapped formats.
- If you create a Spotfire S+ graphics file in JPEG, TIFF, or BMP format, the resulting image can have poor resolution and appear badly pixilated. Enlarging a bitmapped image

produces additional image deterioration: the resolution is not increased, but merely enlarged so that you can see more of the individual pixels. Instead, try exporting your Spotfire S+ graphics to a Windows Metafile (WMF), Enhanced Metafile (EMF), or Encapsulated Postscript (EPS) format. Both of these formats support vector graphics, and will retain all of the information in your Spotfire S+ plots.

Overall, the 16-bit WMF and 32-bit EMF formats are the best reproduction quality format, because they are the only formats drawn from the original graphics primitives.

To access resolution/dpi information from the graphics file:

Table 7.4: *Finding resolution/dpi information for a graphics file.*

Format	Menu command	Resolution Details
EPS	Media ► Display Settings	
JPG	File ► Properties	Default resolution is 300 pixels/inch. Default width is 520 pixels; height is 390 pixels.
PNG	File ► Properties	Default resolution is 97 pixels/inch. Default width is 520 pixels; height is 390 pixels.
TIF	View ► Page Properties	Default resolution is 72 DPI. Default length is 390 pixels; width is 520 pixels.

THE PDF.GRAPH DEVICE

Portable Document File (PDF) is a popular electronic publishing format closely related to PostScript. You can create PDF graphics files in Spotfire S+ using the `pdf.graph` graphics device.

The `pdf.graph` arguments and their defaults are:

```
pdf.graph(file, horizontal = F, width = 0, height = 0,
  pointsize = 14, font = 1, color = T,
  colorspec = if(color) pdf.colors
  else pdf.grays, colormap = colorspec$colormap,
  text.colors = colorspec$text.colors,
  polygon.colors = colorspec$polygon.colors,
  line.colors = colorspec$line.colors,
  image.colors = colorspec$image.colors,
  background.color = colorspec$background.color,
  region = c(0, 0, paper), paper = pdf.paper,
  object, command)
```

For details about these arguments, refer to the `pdf.graph` online help. For information about setting default values for these arguments, refer to the `par` online help.

Creating PDF Files

You can create a PDF version of your plot simply by calling `pdf.graph` with the desired output file name. For example:

```
> pdf.graph("mygraph.pdf")
> plot(corn.rain, corn.yield, main="Another corny plot")
> dev.off()
```

Once you have created a PDF file, you can view it using the Adobe Acrobat Reader (available at <http://www.adobe.com>).

pdf.graph Settings

You can set the following for the `pdf.graph` device:

- symbols
- color
- font (including point size)
- line types

PDF Symbols

The `pch` argument specifies the number of a plotting symbol to be drawn when plotting points. The following table lists the available values for the `pch` argument and their corresponding plotting symbol.

Table 7.5: *PDF symbols.*

pch argument	Symbol
0	square (◻)
1	circle (○)
2	triangle (Δ)
3	cross/plus sign (⊕)
4	X (X)
5	diamond (◇)
6	inverted triangle (∇)
7	steering wheel (⊗)
8	8-point star/asterisk (✳)
9	filled diamond with white X (◆)
10	circle with plus sign (⊕)
11	Star of David (☆)
12	16-point star/asterisk (✴)
13	circle with X (⊗)

Table 7.5: PDF symbols. (Continued)

pch argument	Symbol
14	filled star (5-point) (★)
15	filled square (■)
16	filled circle (●)
17	filled triangle (▲)
18	filled diamond (◆)

The symbols that appear when you set `pch=0:18` are drawn from the character set in the Symbol and ZapfDingbats fonts. The symbols do not look exactly like the marks described in the `points` online help example, but resemble them where possible.

Hint

Setting the `pch` argument to one of the numbers 32 through 126 yields the 95 ASCII characters, from space through tilde.

Example:

```
pdf.graph("test.pdf", horizontal=TRUE)
plot(0:18, 0:18, type="n")
points(0:18, 0:18, pch=0:18, cex=3)
dev.off()
```

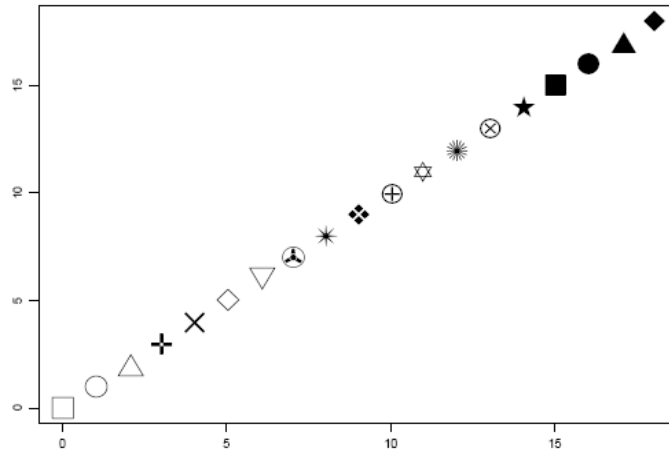


Figure 7.4: *PDF symbols.*

pdf.graph Colors

By default, Spotfire S+ uses a global color palette and global image color palette to map color indexes to specific colors. Previous Spotfire S+ versions performed color mapping on a device-specific basis. For details about the global palettes, see the section Global Color Palette on page 9.

For Spotfire S+ users who have developed code that makes extensive use of the device-specific palettes, the `use.device.palette` function is available for enabling backward compatibility. For usage details, see the section Backward Compatibility on page 24.

For information about device-specific color controls for `pdf.graph`, see the section `pdf.graph Colors` on page 397.

pdf.graph Fonts

The following PDF fonts are supported in Spotfire S+. To use a font, specify its corresponding number for the `font` argument.

Table 7.6: *PDF fonts.*

Argument Value	Font
1	Helvetica (The default.)
2	Courier

Table 7.6: PDF fonts. (Continued)

Argument Value	Font
3	Times-Roman
4	Helvetica-Oblique
5	Helvetica-Bold
6	Helvetica-BoldOblique
7	Courier-Oblique
8	Courier-Bold
9	Courier-BoldOblique
10	Times-Italic
11	Times-Bold
12	Times-BoldItalic
13	Symbol (contains Greek letters and math symbols)
14	AvantGarde-Book
15	AvantGarde-BookOblique
16	AvantGarde-Demi
17	AvantGarde-DemiOblique
18	Bookman-Demi
19	Bookman-DemiItalic

Table 7.6: PDF fonts. (Continued)

Argument Value	Font
20	Bookman-Light
21	Bookman-LightItalic
22	Helvetica-Narrow
23	Helvetica-Narrow-Bold
24	Helvetica-Narrow-BoldOblique
25	Helvetica-Narrow-Oblique
26	NewCenturySchlbk-Roman
27	NewCenturySchlbk-Bold
28	NewCenturySchlbk-Italic
29	NewCenturySchlbk-BoldItalic
30	Palatino-Roman
31	Palatino-Bold
32	Palatino-Italic
33	Palatino-BoldItalic
34	ZapfChancery-MediumItalic
35	ZapfDingbats (contains plotting symbols)

Note

Font size is determined by the `pdf.graph` argument `pointsize`, which is 14 by default. It is the size in 1/72s of inch of text when `par("cex")` is 1.

Example:

```
pdf.graph("test.pdf", horizontal=TRUE)
plot(1:35, 1:35, type="n")
text(1:35, 1:35, "qwerty", font=1:35)
dev.off()
```

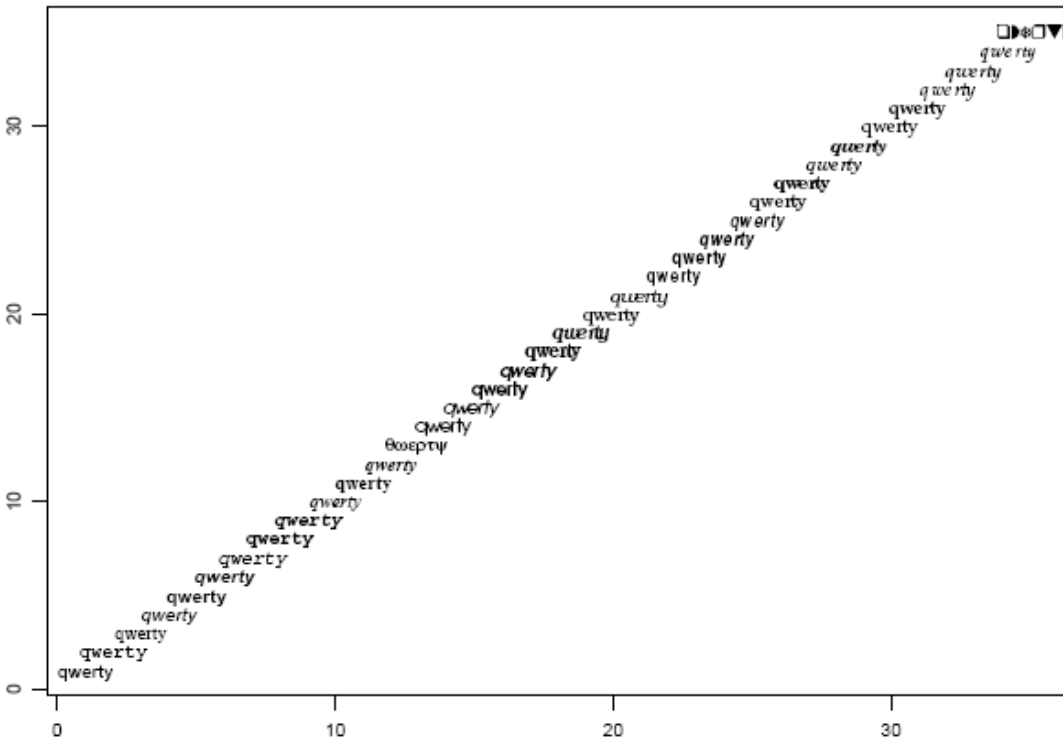


Figure 7.5: *PDF fonts.*

pdf.graph Line Types

The line type is controlled by the `lty` argument. The following line types are supported for PDF.

Table 7.7: *PDF line types.*

Argument Value	Description
1	solid
2	dotted
3	short dashed
4	medium dashed
5	long dashed
6	long dashed dotted
7	medium dashed dotted
8	dotted with more space
9	medium-short dashed (between 3 and 4)
10	“long dashed (very similar to 5, but with a little more space)”

Example:

```
pdf.graph("test.pdf", horizontal=TRUE)
plot(1:10, 1:10, type="n")
for(i in 1:10) lines(c(1,10), c(i,i), lty=i)
dev.off()
```

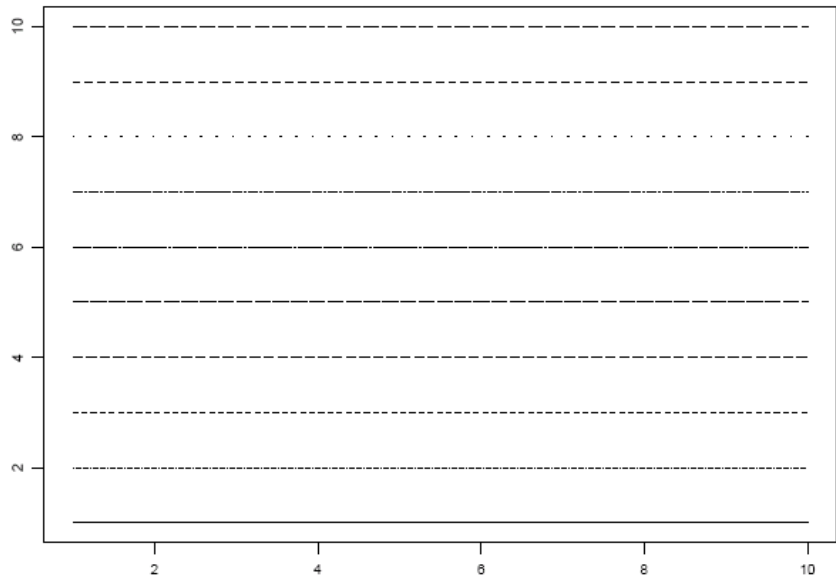


Figure 7.6: *PDF line types.*

THE WMF.GRAPH AND EMF.GRAPH DEVICES

Windows Metafile (WMF) and Enhanced Metafile (EMF) are popular graphics formats for importing graphics into Windows-based programs such as Microsoft Word and Excel. Windows Metafile is the standard 16-bit format, while Enhanced Metafile is the 32-bit format.

You can create WMF and EMF graphics files in Spotfire S+ using the `wmf.graph` and `emf.graph` graphics devices, respectively.

The `wmf.graph` arguments and their defaults are:

```
wmf.graph(file, horizontal = F, width = 7, height = 5.4,
  pointsize = 14, fonts = character(0), color = T,
  colorspec = if(color) pdf.colors else pdf.grays,
  colormap = colorspec$
  colormap, text.colors = colorspec$text.colors,
  polygon.colors = colorspec$polygon.colors,
  line.colors = colorspec$line.colors,
  image.colors = colorspec$image.colors,
  background.color = colorspec$background.color,
  region = c(0, 0, paper), paper = pdf.paper, command,
  paint.background = T, line.width.factor = 15,
  text.line.spacing.factor = 0.8)
```

The `emf.graph` arguments and their defaults are:

```
emf.graph(file, horizontal = F, width = 7, height = 5.4,
  pointsize = 14, fonts = character(0), color = T,
  colorspec = if(color) pdf.colors else pdf.grays,
  colormap = colorspec$colormap,
  text.colors = colorspec$text.colors,
  polygon.colors = colorspec$polygon.colors,
  line.colors = colorspec$line.colors,
  image.colors = colorspec$image.colors,
  background.color = colorspec$background.color,
  region = c(0, 0, paper), paper = pdf.paper, command,
  paint.background = T, line.width.factor = 15,
  text.line.spacing.factor = 0.8)
```

For details about these arguments, refer to the `wmf.graph` and `emf.graph` online help. For information about setting default values for these arguments, refer to the `par` online help.

Creating WMF or EMF Graphics

You can create a WMF or EMF graphics file simply by calling `wmf.graph` or `emf.graph` with the desired output file name. For example:

```
> wmf.graph("mygraph.wmf")
> plot(corn.rain, corn.yield, main="Another corny plot")
> dev.off()
```

or

```
> emf.graph("mygraph.emf")
> plot(corn.rain, corn.yield, main="Another corny plot")
> dev.off()
```

wmf.graph and emf.graph Settings

You can set the following for the `wmf.graph` or `emf.graph` devices:

- symbols
- color
- font (including point size)
- line types

wmf.graph and emf.graph Symbols

The `pch` argument specifies the number of a plotting symbol to be drawn when plotting points. Table 7.8 lists the available values for the `pch` argument and their corresponding plotting symbol. Symbols produced are the same as those produced for `java.graph`. The symbols for these argument values are shown in Figure 7.7.

Table 7.8: *WMF and EMF symbols.*

Argument Value	Description
0	square
1	circle
2	triangle
3	cross
4	X
5	diamond

Table 7.8: *WMF and EMF symbols. (Continued)*

Argument Value	Description
6	inverted triangle
7	square + X
8	X + cross
9	diamond + cross
10	circle + cross
11	triangle + inverted triangle
12	square + cross
13	circle + X
14	square + triangle
15	filled square
16	filled circle
17	filled triangle
18	filled diamond

These are the same marks described in the `points` online help.

Assigning the values 32 through 126 to the argument `pch` yields the 95 ASCII characters, from space through tilde.

WMF Example

```
wmf.graph("test.wmf", horizontal=TRUE)
plot(0:18, 0:18, type="n")
points(0:18, 0:18, pch=0:18, cex=3)
```

`dev.off()`

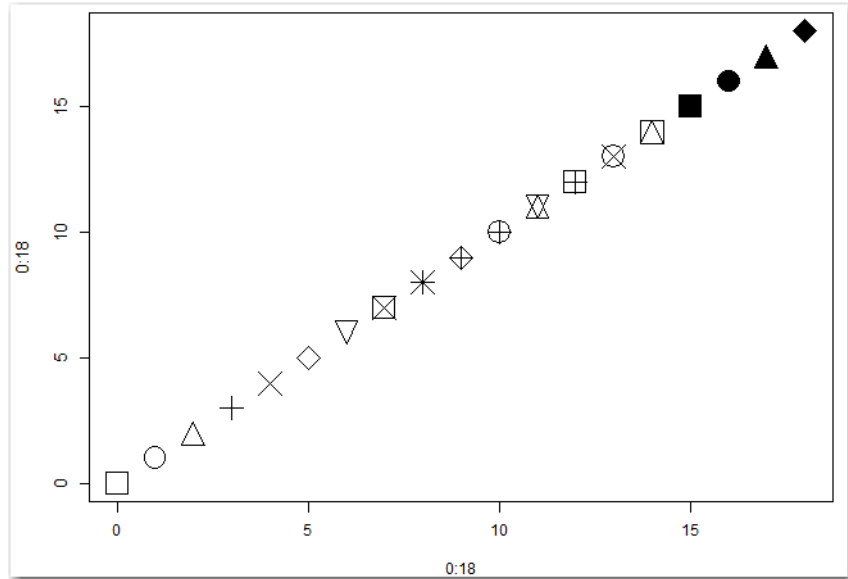


Figure 7.7: *WMF and EMF symbols.*

wmf.graph and emf.graph Colors

By default, Spotfire S+ uses a global color palette and global image color palette to map color indexes to specific colors. Previous Spotfire S+ versions performed color mapping on a device-specific basis. For details about the global palettes, see the section Global Color Palette on page 9.

For Spotfire S+ users who have developed code that makes extensive use of the device-specific palettes, the `use.device.palette` function is available for enabling backward compatibility. For usage details, see the section Backward Compatibility on page 24.

For information about device-specific color controls for the `wmf.graph` or `emf.graph` device, see the section `wmf.graph` and `emf.graph` Colors on page 401.

wmf.graph and emf.graph Fonts

The following WMF and EMF fonts are supported in Spotfire S+. To use a font, specify its corresponding number for the font argument. For example, specify font=3 for Courier New.

Table 7.9: *WMF and EMF fonts.*

Argument Value	Font name
1 or 0	Arial (The default)
2	Times New Roman
3	Courier New
4	Tahoma
5	Modern
6	MS Sans Serif
7	Script
8	Symbol

Example:

```
wmf.graph("test.wmf", horizontal=TRUE)
plot(1:8, 1:8, type="n")
text(1:8, 1:8, "qwerty", font=1:8)
dev.off()
```

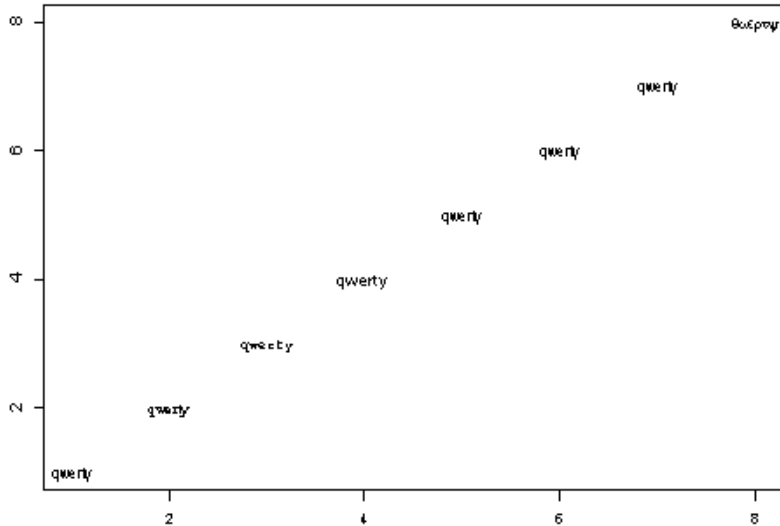


Figure 7.8: *WMF and EMF fonts.*

To use a font that is not listed above, specify the Windows typeface name for font in `wmf.graph` and `emf.graph`. For example:

```
emf.graph("test.emf", font="Americana ExBld")
plot(1:8, 1:8)
text(3, 7, "americana bold")
dev.off()
```

Note

Font size is determined by the `wmf.graph` or `emf.graph` argument `pointsize`, which is 14 by default. It is the size in 1/72 of inch of text when `par("cex")` is 1.

wmf.graph and emf.graph Line Types

The line type is controlled by the `lty` argument. The following line types are supported for the `wmf.graph` and `emf.graph` devices.

Table 7.10: *WMF and EMF line types*

Argument value	Line type
1	solid line
2	short dashed (dotted)
3	long dashed
4	dots and dashes
5	dash dot dot

Lines are drawn `par("lwd")*line.width.factor/1000` inches wide. By default, `line.width.factor=15` and `par("lwd")` is 1, so lines are drawn $3/200$ (or 0.015) inches wide.

EMF Example:

```
emf.graph("test.emf", horizontal=TRUE)
plot(1:5, 1:5, type="n")
for(i in 1:5) lines(c(1,5), c(i,i), lty=i)
dev.off()
```

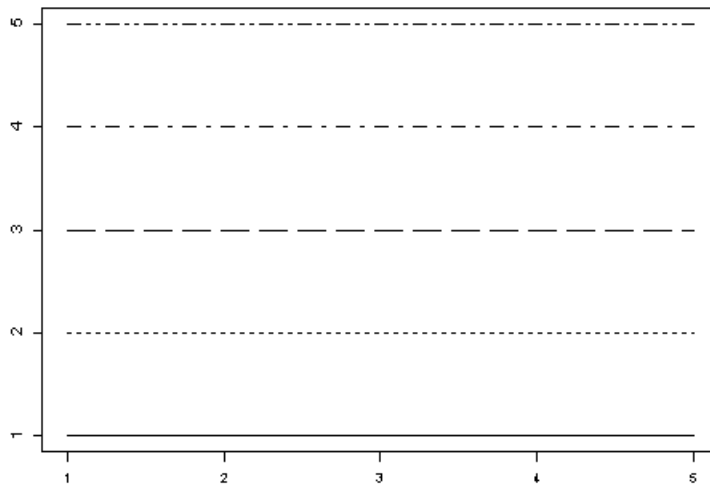


Figure 7.9: *WMF and EMF line types*

THE POSTSCRIPT DEVICE

You can create PostScript graphics files in Spotfire S+ using the `postscript` graphics device.

The `postscript` arguments and their defaults are:

```
postscript(file = NULL, width = -1, height = -1,  
  append = F, onefile = T, print.it = NULL,  
  color.p = F, ..., old.style = F)
```

For details about these arguments, refer to the `postscript` online help. For information about setting default values for these arguments, refer to the `par` online help.

postscript **Device Settings**

You can set the following for the `postscript` device:

- symbols
- color
- font (including point size)
- line types

postscript **Symbols**

The `pch` argument specifies the number of a plotting symbol to be drawn when plotting points. Table 7.11 lists available values for the `pch` argument and their corresponding plotting symbol.

The symbol for each argument value is shown in Figure 7.7.

Table 7.11: *PostScript Symbols.*

Argument value	Symbol
0	square
1	circle
2	triangle
3	cross
4	X

Table 7.11: *PostScript Symbols. (Continued)*

Argument value	Symbol
5	diamond
6	inverted triangle
7	square + X
8	X + cross
9	diamond + cross
10	circle + cross
11	triangle + inverted triangle
12	square + cross
13	circle + X
14	square + triangle
15	filled square
16	filled circle
17	filled triangle
18	filled diamond

These are the same basic marks that are described in the `points` online help.

Assigning the values 32 through 126 to the argument `pch` yields the 95 ASCII characters, from space through tilde.

Example

```
postscript("test.eps", horizontal=TRUE)
plot(0:18, 0:18, type="n")
points(0:18, 0:18, pch=0:18, cex=3)
dev.off()
```

postscript Colors By default, Spotfire S+ uses a global color palette and global image color palette to map color indexes to specific colors. Previous Spotfire S+ versions performed color mapping on a device-specific basis. For details about the global palettes, see the section Global Color Palette on page 9.

For Spotfire S+ users who have developed code that makes extensive use of the device-specific palettes, the `use.device.palette` function is available for enabling backward compatibility. For usage details, see the section Backward Compatibility on page 24.

For information about device-specific color controls for the `postscript` device, see the section `postscript Colors` on page 403.

postscript Fonts The `postscript` device supports the fonts listed in Table 7.12 on page 393.

The `fonts` argument of the `ps.options` function is a character object enumerating the fonts you want available in the `postscript` device.

The `font` argument of the `ps.options` function is the number that specifies the default font for the text. A negative number selects the font in its outline form. The default value is 1, specifying the first font in `ps.options$fonts`. The `font` argument can also be specified in the plotting command.

Font size is determined by the `ps.options` argument `pointsize`, which is 14 by default. It is the base size of text, in points (one point is equal to 1/72 inches). If `par("cex")` is 1, text appears in the size specified by `pointsize`.

Refer to the `ps.options` online help for more information.

To use a font, specify its corresponding number for the font argument. For example, specify font=3 for Times-Roman.

Table 7.12: *PostScript fonts.*

Argument Value	Font
1	Helvetica
2	Courier
3	Times-Roman
4	Helvetica-Oblique
5	Helvetica-Bold
6	Helvetica-BoldOblique
7	Courier-Oblique
8	Courier-Bold
9	Courier-BoldOblique
10	Times-Italic
11	Times-Bold
12	Times-BoldItalic
13	Symbol
14	AvantGarde-Book
15	AvantGarde-BookOblique
16	AvantGarde-Demi

Table 7.12: *PostScript fonts. (Continued)*

Argument Value	Font
17	AvantGarde-DemiOblique
18	Bookman-Demi
19	Bookman-DemiItalic
20	Bookman-Light
21	Bookman-LightItalic
22	Helvetica-Narrow
23	Helvetica-Narrow-Bold
24	Helvetica-Narrow-BoldOblique
25	Helvetica-Narrow-Oblique
26	NewCenturySchlbk-Roman
27	NewCenturySchlbk-Bold
28	NewCenturySchlbk-Italic
29	NewCenturySchlbk-BoldItalic
30	Palatino-Roman
31	Palatino-Bold
32	Palatino-Italic

Table 7.12: *PostScript fonts. (Continued)*

Argument Value	Font
33	Palatino-BoldItalic
34	ZapfChancery-MediumItalic
35	ZapfDingbats

Example:

```

postscript("test.eps", horizontal=TRUE)
plot(1:35, 1:35, type="n")
text(1:35, 1:35, "qwerty", font=1:35)
dev.off()

```

postscript Line Types

The line type is controlled by the `lty` argument. The `postscript` device supports the following line types.

Table 7.13: *PostScript line types.*

Argument value	Line type description
1	solid
2	dotted
3	short dashed
4	medium dashed
5	long dashed
6	long dashed dotted
7	medium dashed dotted
8	dotted with more space

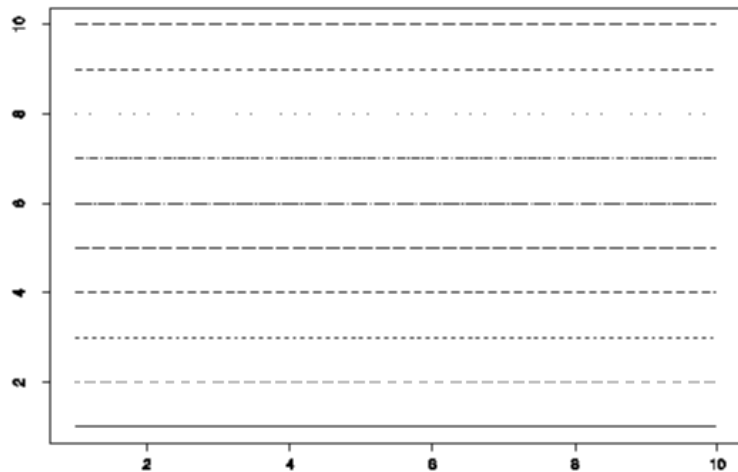
Table 7.13: *PostScript line types. (Continued)*

Argument value	Line type description
9	medium-short dashed (between 3 and 4)
10	“long dashed (very similar to 5, but with a little more space)”

Line width is specified by the `lwd` parameter and is interpreted in units of 1/36 inches. The default `lwd=1` line is 2 points wide. A line width of 0 draws the thinnest possible line on the device.

Example:

```
postscript("test.eps", horizontal=TRUE)
plot(1:10, 1:10, type="n")
for(i in 1:10) lines(c(1,10), c(i,i), lty=i)
dev.off()
```

**Figure 7.10:** *postscript line types*

DEVICE-SPECIFIC COLOR SPECIFICATION

Most Spotfire S+ users should disregard this section and use the default *global color palette* and *global image color palette* described in the section Global Color Palette on page 9.

For Spotfire S+ users who have developed code that makes extensive use of the device-specific palettes, the `use.device.palette` function provides a means of disabling the global palettes and enabling backward compatibility. For usage details, see the section Backward Compatibility on page 24.

Note

The information in this “Device-Specific Color Specification” section applies *only* if you are using Spotfire S+ in the backward compatible `use.device.palette(T)` mode. For usage details, see the section Backward Compatibility on page 24.

When using Spotfire S+ in the default `use.device.palette(F)` mode, disregard this information and refer to the section Color Specification on page 3 for information about working with colors.

pdf.graph Colors

The `pdf.graph`, `wmf.graph`, and `emf.graph` devices use `pdf.colors$colormap` for their color mapping. These devices do not control color by name, but by the argument `colorspec`, which specifies the components from the arguments `colormap`, `line.colors`, `text.colors`, `polygon.colors`, and `image.colors`.

The following arguments in the `pdf.graph`, `wmf.graph`, and `emf.graph` functions control color:

Table 7.14: *pdf.graph color arguments.*

Argument	Description
<code>color</code>	If TRUE (the default) use colors; if FALSE use grayscale. This argument sets the default for <code>colorspec</code> to either <code>pdf.colors</code> or <code>pdf.grays</code> .

Table 7.14: *pdf.graph* color arguments. (Continued)

Argument	Description
<code>colormap</code>	A list containing a colormap and four vectors of indices into that colormap: one vector each for use when drawing lines, text, polygons, and images. The components must be named <code>colormap</code> , <code>line.colors</code> , <code>text.colors</code> , <code>polygon.colors</code> , and <code>image.colors</code> . You can give those components as individual arguments to <code>pdf.graph</code> ; however, doing this overrides the corresponding component of <code>colormap</code> . See the corresponding argument descriptions for more information. See the datasets <code>pdf.grays</code> and <code>pdf.colors</code> for examples.
<code>colormap</code>	For grayscale, a vector of numbers between 0 (black) and 1 (white). For color, a three-column by <code>ncolor</code> -row matrix of numbers between 0 and 1. Each row represents a color and its entries give the intensity of its red, green, and blue components. (Instead of this matrix, you can also use an <code>ncolor</code> long list of three long vectors.)
<code>text.colors</code>	Indexes used in the <code>colormap</code> , giving the colors for text. The color in: <code>colormap[text.colors[par(“col”)]]</code> (or corresponding row of <code>colormap</code> if it is a matrix) will be used for text. Typically, <code>text.colors</code> , <code>line.colors</code> , and <code>polygon.colors</code> point to contrasting colors.
<code>polygon.colors</code>	Indexes used in <code>colormap</code> for filled polygon colors.
<code>line.colors</code>	Indexes used in <code>colormap</code> for line colors.
<code>image.colors</code>	Indexes used in <code>colormap</code> for image colors. Typically, <code>image.colors</code> points to a smoothly changing sequence of colors.
<code>background.color</code>	index into <code>colormap</code> for background color.

To display the red/green/blue (RGB) values for the default background, at the Spotfire S+ prompt in the **Commands** window, type:

```
pdf.colors$colormap[pdf.colors$background.color,]
```

Spotfire S+ returns the following:

```
[1] 1 1 1 #white
```

To display RGB values for the default line, polygon, and text colors, at the Spotfire S+ prompt in the **Commands** window, type:

```
pdf.colors$colormap[pdf.colors$line.colors,]
```

```
[1,] 0.0000000 0.0000000 0.0000000 #black
[2,] 0.0000000 0.3725490 0.7450980 #blue
[3,] 0.7450980 0.0000000 0.3725490 #fuschia
[4,] 0.0000000 0.7450980 0.0000000 #green
[5,] 1.0000000 0.5019608 0.0000000 #orange
[6,] 0.0000000 0.5019608 1.0000000 #light blue
[7,] 0.5019608 0.2509804 0.0000000 #brown
[8,] 0.7529412 0.0000000 0.0000000 #red
[9,] 0.7215686 1.0000000 1.0000000 #robin's egg blue
[10,] 1.0000000 0.7647059 1.0000000 #light lavender
[11,] 0.7843137 1.0000000 0.7843137 #mint green
[12,] 1.0000000 0.8196078 0.5607843 #tan
[13,] 0.6627451 0.8862745 1.0000000 #sky blue
[14,] 1.0000000 1.0000000 0.7647059 #pale yellow
[15,] 1.0000000 0.5490196 0.5411765 #pale orange
[16,] 0.4313725 0.4313725 0.3921569 #gray
```

To create a PDF that displays the colors, type the following in the **Commands** window of Spotfire S+:

```
# Test graph
pdf.graph("test.pdf", horizontal=TRUE)
plot(1:16, 1:16, type="n")
for(i in 1:16) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

Customizing Colors

Suppose you want to use PDF or WMF colors similar to the default colors in a Spotfire S+ (Windows) graph. The graphsheets window in Spotfire S+ has its own color map. Both maps contain RGB values; however, the pdf.colors map values range from 0 to 1, where the Spotfire S+ graph sheet map ranges from 0 to 255.

To get PDF colors similar to default Spotfire S+ colors, modify pdf.colors as follows:

```
# Create a vector of RGB values:
color.vec <- c(0,0,0,255,0,0,255,102,0,255,102,153,255,
              204,153,255,255,0,204,255,102,102,255,0,0,102,0,0,
              51,51,102,0,0,204,0,153,255,102,204,255,102,102,204,
              0,0,102,0,153,255,255,255)

# Create a matrix of RGB values from the vector:
color.mat <- matrix(color.vec, ncol=3, byrow=T)

# Divide by 255 so values range from 0 to 1 for use
#in the pdf.colors map:
color.mat <- color.mat/255

# Modify pdf.colors:
pdf.colors <- pdf.colors
pdf.colors$colormap <- color.mat
pdf.colors$image.colors <-
c(1,2,6,13,9,11,4,16,7,5,12,14,15,8,10,3)
pdf.colors$line.colors <- 1:16
pdf.colors$polygon.colors <- 1:16
pdf.colors$text.colors <- 1:16
pdf.colors$background.color <- 17

# Test graph:
pdf.graph("test.pdf", horizontal=TRUE)
plot(1:16, 1:16, type="n")
for(i in 1:16) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

When you open **test.pdf**, the resulting graph appears as follows:

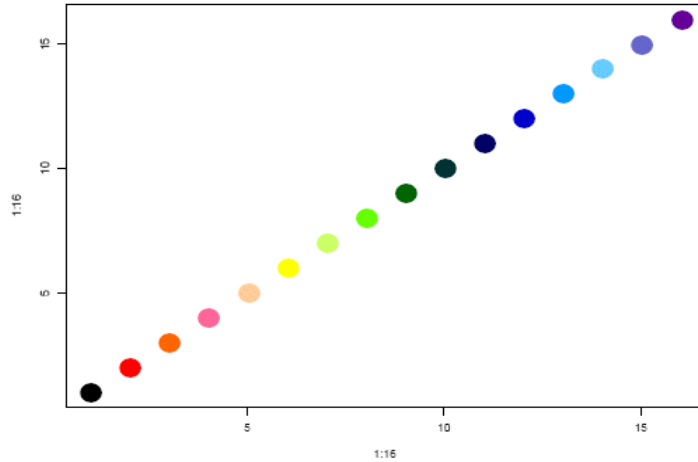


Figure 7.11: *PDF color example.*

Alternatively, it's possible to specify the color arguments in `pdf.graph`. For example:

```
pdf.graph("test.pdf", horizontal=TRUE,
  colormap=color.mat, line.colors=1:16, text.colors=1:16,
  polygon.colors=1:16, image.colors=1:16,
  background.color=17)
plot(1:16, 1:16, type="n")
for(i in 1:16) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

wmf.graph and emf.graph Colors

The `wmf.graph` and `emf.graph` device uses the same color map as the `pdf.graph` device. For information about setting colors for `wmf.graph` and `emf.graph`, see the section `pdf.graph Colors` on page 397.

wmf.graph Color Example

Suppose you want to have `wmf.graph` colors similar to the default colors in a Spotfire S+ (Windows) graph. The graphsheet window in Spotfire S+ has its own color map. Both maps contain RGB values; however, the `pdf.colors` map values range from 0 to 1 whereas the Spotfire S+ graphsheet map ranges from 0 to 255.

```
# Create a vector of values from the Graph Sheet Color
# Table (in this case the default colors):
color.vec <-
  c(0,0,0,0,64,128,128,0,64,0,128,0,255,128,0,0,128,255,
    128,64,0,192,0,0,198,255,255,255,195,255,200,255,
    200,255,209,143,169,226,255,255,255,195,255,140,138,
    110,110,100,255,255,255)

# Create a matrix of RGB values from the vector:
color.mat <- matrix(color.vec, ncol=3, byrow=T)

# Divide by 255 so values range from 0 to 1 for use in
# the pdf.colors map:
color.mat <- color.mat/255

# Modify pdf.colors:
pdf.colors <- pdf.colors
pdf.colors$colormap <- color.mat
pdf.colors$image.colors <-
  c(1,2,6,13,9,11,4,16,7,5,12,14,15,8,10,3)
pdf.colors$line.colors <- 1:16
pdf.colors$polygon.colors <- 1:16
pdf.colors$text.colors <- 1:16
pdf.colors$background.color <- 17

# Test graph:
wmf.graph("test.wmf", horizontal=TRUE)
plot(1:16, 1:16, type="n")
for(i in 1:16) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

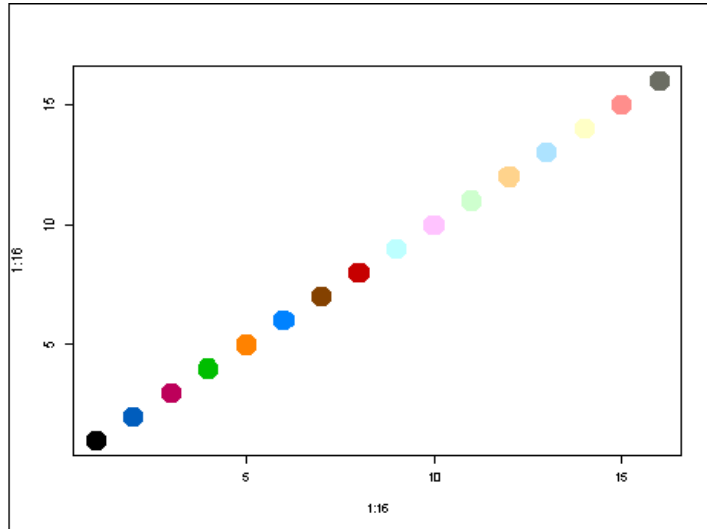


Figure 7.12: *wmf.graph.colors*.

Alternatively, you can specify the color arguments in `wmf.graph`. See the example in the section Customizing Colors on page 400.

postscript Colors

To display PostScript default colors, in the Spotfire S+ **Commands** window, type the following:

```
ps.options()$colors
[1] 0.0 0.6 0.3 0.9 0.4 0.7 0.1 0.5 0.8 0.2
```

This is a vector, defining gray levels (0 = black to 1 = white).

To display the default background color, type the following:

```
ps.options()$background
[1] -1 #no color
```

**Managing
postscript Colors**

The following arguments to the `ps.options` function are specific to color:

Table 7.15: *PostScript color options.*

Argument	Description
background	A numeric object specifying a background color to use. If background is a single positive integer, it is treated as an index into the <code>colors</code> argument. If it is a negative number, then there is no background. Otherwise, it is assumed to be a single explicit color specification, as in the <code>colors</code> argument. The default is -1 (no background).
colors	A numeric object giving an assignment of colors to color numbers. Using the default <code>setColor</code> specification, there are two possibilities. If <code>colors</code> is a vector, it is used to define gray levels (0=black to 1=white) beginning with color number 1. If <code>colors</code> is a three-column matrix, it is used to define colors in the RGB model (see the <code>postscript</code> online help), one per row, beginning with color number 1. <code>ps.colors.rgb</code> can be subsetted to provide a value for this argument. Also, the function <code>ps.hsb2rgb</code> can be used to convert colors from the HSB model to the RGB model equivalents.
image.colors	Numeric objects giving an assignment of colors to color numbers. These colors are used as fill colors for use with the <code>image</code> function. Refer to the <code>colors</code> component for specification details.

Here are examples of specifying colors for PostScript files:

- Create a vector of color names from the `ps.colors.rgb` matrix that you can use when specifying colors in `ps.options`.

```
color.vec <-
  c("black","blue","green","cyan","gold","purple",
    "orange","red","orchid","gray")
ps.options(colors=ps.colors.rgb[color.vec,])
postscript("test.eps", horizontal=TRUE)
plot(1:10, 1:10, type="n")
```

```
for(i in 1:10) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

- Reset to the default black and white with the following command:

```
ps.options(reset=T)
```

- Alternatively, you can specify colors in the PostScript command:

```
postscript("test.eps", horizontal=TRUE,
           colors=ps.colors.rgb[color.vec,])
plot(1:10, 1:10, type="n")
for(i in 1:10) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

By following this technique, you do not need to reset `ps.options`.

- Suppose you want to have postscript colors similar to the default colors in a Spotfire S+ (Windows) graph. The graph sheet window in Spotfire S+ has its own color map with RGB values that range from 0 to 255. The PostScript colors matrix of RGB values should range from 0 to 1 (as you can see with `ps.colors.rgb`).

To get postscript colors similar to default Spotfire S+ colors, you can create a color matrix as follows:

```
# Create a vector of values from the Graph Sheet
# Color Table (in this case the default colors):
color.vec <-
  c(0,0,0,0,64,128,128,0,64,0,128,0,255,128,0,0,
    128,255,128,64,0,192,0,0,198,255,255,255,195,
    255,200,255,200,255,209,143,169,226,255,255,
    255,195,255,140,138,110,110,100,255,255,255)
```

```
# Create a matrix of RGB values from the vector:
color.mat <- matrix(color.vec, ncol=3, byrow=T)
```

```
# Divide by 255 so values range from 0 to 1 for use
# in the colors argument:
color.mat <- color.mat/255
```

```
# Test graph:
postscript("test.eps", horizontal=TRUE,
          colors=color.mat)
plot(1:16, 1:16, type="n")
for(i in 1:16) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

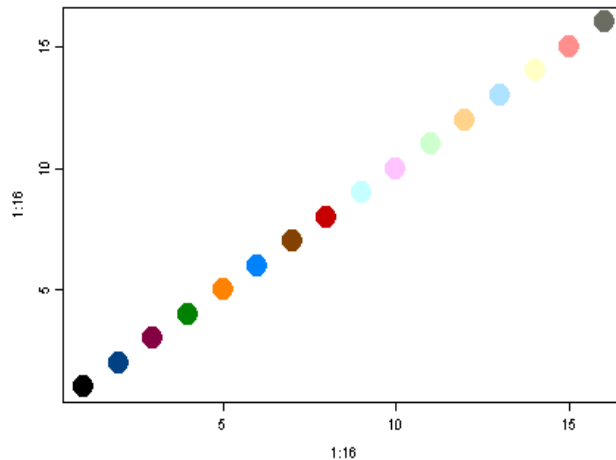


Figure 7.13: *postscript colors.*

You can set the background color by specifying the background argument. Here are a couple different ways:

- Set background to a positive integer so it is an index into the colors argument.

```
color.vec <-
  c("black","blue","green","cyan","gold","purple",
    "orange","red","orchid","gray","light blue")
ps.options(colors=ps.colors.rgb[color.vec],
          background=11) #index for light blue
postscript("test.eps", horizontal=TRUE)
plot(1:10, 1:10, type="n")
for(i in 1:10) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

- Set background to a single explicit color specification. See Figure 7.14.


```
ps.options(colors=ps.colors.rgb[color.vec,],
  background=c(0.9019608,0.9019608,0.9803922)) #rgb for
lavender
postscript("test.eps", horizontal=TRUE)
plot(1:10, 1:10, type="n")
for(i in 1:10) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

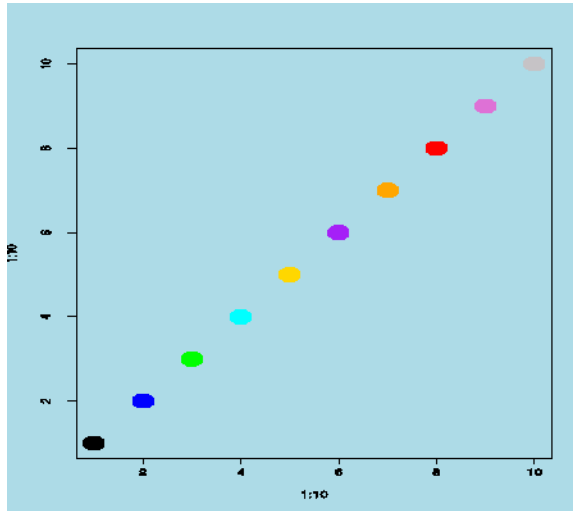


Figure 7.14: *postscript* color example.

You can also use `ps.colors.rgb` with `pdf.graph`, `wmf.graph`, and `emf.graph`. For example:

```
color.vec <-
  c("black","blue","green","cyan","gold","purple",
    "orange","red","orchid","gray","white")
wmf.graph("test.wmf", horizontal=TRUE,
  colormap=ps.colors.rgb[color.vec,], line.colors=1:10,
  text.colors=1:10,polygon.colors=1:10, image.colors=1:10,
  background.color=11)
plot(1:10, 1:10, type="n")
for(i in 1:10) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

java.graph Colors

To set the color scheme for a Java-enabled Spotfire S+ Graphlet, use the argument `colorscheme` in the function `java.graph`.

To get the list of RGB colors for `java.graph`, in the Spotfire S+ **Commands** window, type:

```
java.colorscheme.default$palette

      [,1] [,2] [,3]
[1,]    0    0    0 #black
[2,]    0   95  190 #navy
[3,]  190    0   95 #dark magenta
[4,]    0  190    0 #green
[5,]  255  128    0 #orange
[6,]    0  128  255 #blue
[7,]  128   64    0 #brown
[8,]  192    0    0 #red
[9,]  184  255  255 #cyan
[10,] 255  195  255 #violet
[11,] 200  255  200 #pale green
[12,] 255  209  143 #light orange
[13,] 169  226  255 #light blue
[14,] 255  255  195 #light yellow
[15,] 255  140  138 #light coral
[16,] 110  110  100 #gray
```

To get the default background color, in the **Commands** window, type:

```
java.colorscheme.default$background

[1] 255 255 255 #white
```

The `colorscheme` argument of `java.graph` contains the following predefined color scheme variables:

- `java.colorscheme.default`
- `java.colorscheme.standard`
- `java.colorscheme.trellis`
- `java.colorscheme.trellis.black.on.white`
- `java.colorscheme.white.on.black`

- `java.colorscheme.cyan.magenta`
- `java.colorscheme.topographical`
- `java.colorscheme.user.1`
- `java.colorscheme.user.2`

`java.graph` does not control color by name; you must specify its scheme.

`java.colorscheme.default` is almost identical to the default color scheme in the Spotfire S+ for Windows graphsheet.

Each `java.graph` device has a color scheme that determines the color of the background, the colors of lines, text, and so on, and the colors of image bitmaps. Currently there is a single color palette used for lines, text, symbols, and polygons.

Example

```
java.graph(file="test.jpg")
plot(1:16, 1:16, type="n")
for(i in 1:16) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

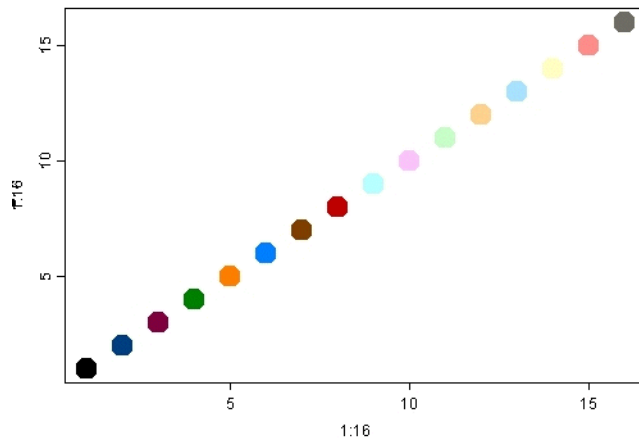


Figure 7.15: *java.graph* default colors.

Example of changing colors:

```
# Create template for new color scheme:
java.colorscheme.new <- java.colorscheme.standard
java.colorscheme.new$name <- "My New Colors"
```

```
# Create a vector of RGB values:
color.vec <-
  c(0,0,0,102,51,204,153,0,51,255,204,51,51,0,255,
    153,102,153,102,255,102,153,204,255,255,0,0,
    255,255,102,0,255,204,153,153,204,0,0,130,
    255,153,204,153,102,51,153,153,153)

# Create a matrix from the vector:
color.mat <- matrix(color.vec, ncol=3, byrow=T)

# Assign RGB matrix to color palette:
java.colorscheme.new$palette <- color.mat

# Test graph:
java.graph(file="test.jpg",
           colorscheme=java.colorscheme.new)
plot(1:16, 1:16, type="n")
for(i in 1:16) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

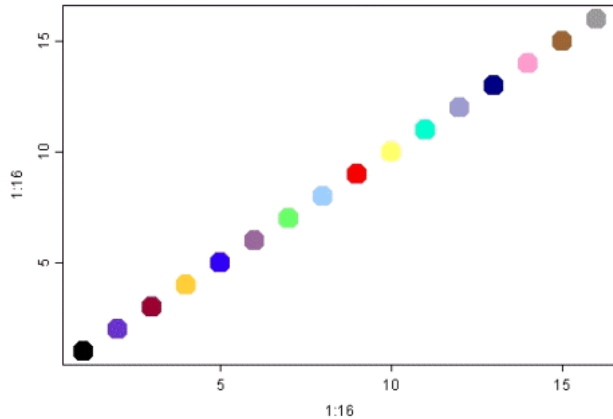


Figure 7.16: *java.graph color change.*

Example with a different background color:

```
java.colorscheme.new$background <- c(230,230,250)
#lavender

java.graph(file="test.jpg",
           colorscheme=java.colorscheme.new)
plot(1:16, 1:16, type="n")
```

```
for(i in 1:16) points(i, i, col=i, cex=3, pch=16)
dev.off()
```

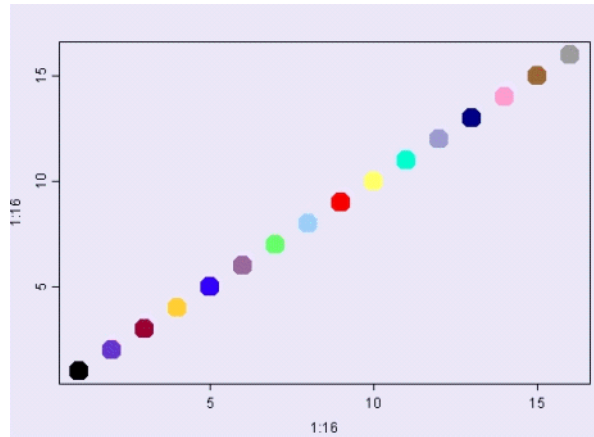


Figure 7.17: `java.graph` background color change.

Using the S-PLUS 6.2 Colorscheme

Each device type has functions that set the colorscheme to the S-PLUS 6.2 setting. Some require resetting for each session and some retain the setting state.

Graphsheet

If you want to use the old color settings for a graphsheet, run the function `stdImageColorSet(blue=F)`. You must run this function during every session, because `stdImageColorSet(blue=T)` is called in the startup file **S.init**. You can remove the call to `stdImageColorSet` in **S.init**; the file is in the top level directory where Spotfire S+ is installed.

java.graph

If you use `java.graph`, the S-PLUS 6.2 colorscheme is available in the list `java.colorscheme.standard.6.2`. Run the following:

```
java.colorscheme.default <- java.colorscheme.standard.6.2
```

All subsequent calls to `java.graph` (where the `colorscheme` argument is not explicitly set) in the current project directory will use the old colorscheme. If you switch to a different project directory, you must recreate the `java.colorscheme.default` object, as shown above.

pdf.graph, wmf.graph, and emf.graph If you use `pdf.graph`, `wmf.graph`, or `emf.graph`, set the `colorespec` argument to `pdf.colors.6.2` or `pdf.grays.6.2` to get the old `colormaps`.

postscript Device You can set the new argument, `old.style=T` to get the old behavior.

A new argument, `color.p` has been added to the PostScript device. Setting `color.p=T` (or just `color=T`) creates color PostScript output using the new `colormaps`. The default is `color.p=F` so that `postscript` continues to produce black-and-white PostScript by default.

The behavior and setting of graphics devices started with the `trellis.device` function has not changed.

Introduction	414
Requirements	416
Examples	416
Creating a Graphlet Data File	418
Interactive File Creation	418
Programmatic File Creation	418
Page Titles and Tags	418
Active Regions	419
Action Strings	422
A Detailed Example	426
Embedding the Graphlet in a Web Page	430
Applet Parameters	431
A Detailed Example (Continued)	435
Using the Graphlet	436

INTRODUCTION

The Spotfire S+ Graphlet is a Java applet that supports displaying and interacting with Spotfire S+ graphics that have been saved in a file. The Graphlet can be embedded in a Web page and viewed with a Web browser. For example, Figure 8.1 shows an example Graphlet displayed in Internet Explorer.

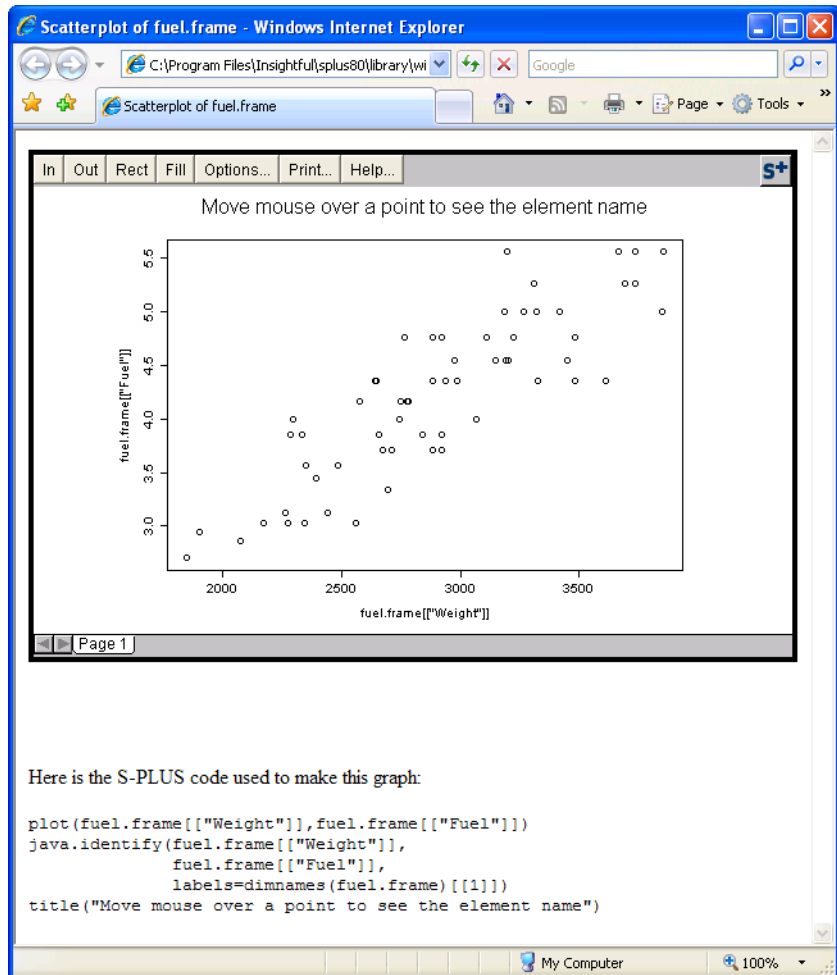


Figure 8.1: An example Graphlet.

Displaying Spotfire S+ graphs with the Graphlet provides several interactive features not available with static graph images such as JPEG or GIF images:

1. The Graphlet displays a graph window with multiple tabbed pages, similar to the Spotfire S+ `java.graph` graphics display. In many cases, it is useful to collect a set of related graphs into a number of pages. The graphics creator can set the titles of the page tabs to meaningful names.
2. The displayed graph can be panned and zoomed to examine details more closely. The graph is represented as a series of graph commands rather than as a bitmap, so a zoomed image is just as sharp as the original size.
3. The graphics creator can define any number of rectangular *active regions* within a page. As you move the mouse over an active region, it is outlined and a label associated with the region is displayed. The label is a string that is used to display additional information for a given element of the graph. For example, active regions can be defined around all of the points in a scatter plot, giving additional information for every point.
4. Each active region may also have an associated *action string*, which specifies what should happen if the mouse is clicked in an active region. Options include switching to another page in the Graphlet, having the Web browser jump to a specified Web page, or popping up a menu for selecting one of multiple such actions. This provides a way to “drill down” and display more information about a particular region of the graph.
5. The *graph coordinates* of the mouse pointer can be displayed as the pointer is moved over the graph. The graph coordinates are the (x,y) coordinates of the mouse position according to the graph axes, rather than just the (x,y) pixel coordinates within the window. Thus, the display can be used to examine the positions of points in a scatter plot or other plot where the coordinates are of interest. If there are multiple graphs in a single page, the graph coordinates of the graph containing the mouse are displayed.

Note that there is no direct link between the Graphlet and Spotfire S+. Using Spotfire S+, a series of graphics commands is stored in a file and the file is closed. Later, the Graphlet reads the file and displays the graphics, perhaps long after the Spotfire S+ session has ended, perhaps by a user over the Web who does not have Spotfire S+.

There are three main steps in creating and using Spotfire S+ Graphlets:

1. Create a Graphlet data file in the SPJ format.
2. Embed the Graphlet in a Web page.
3. View and interact with the Graphlet.

In this chapter, we discuss each of these steps in detail. A *graphics creator* is typically concerned with the first two steps, while an *end user* is concerned only with the third. If you are an end-user who needs to know how to interact with Spotfire S+ Graphlets, you can skip to the section Using the Graphlet (page 436).

Requirements

The Spotfire S+ Graphlet is designed to work with Web browsers that support Java applets with Java version 1.1 or later.

To test whether your browser supports Java applets try viewing the following page, which contains a simple Java applet:

<http://www.insightful.com/products/graphlets/gallery/TestJavaApplet.asp>

If your browser does not support Java applets, install the free Java Runtime Environment available at <http://www.java.com>.

The Graphlet displays traditional Spotfire S+ graphics. For details on graphics commands and options, see the following two chapters:

- Chapter 2, Traditional Graphics
- Chapter 3, Traditional Trellis Graphics

Examples

Example Graphlets are included in the following locations in your Spotfire S+ installation:

- **Windows:** <Spotfire S+ install dir>\library\winspj

- **UNIX: \$SHOME/library/example5/graphlet**
Before attempting to view these files, copy the ***.html** and ***.spj** files, along with **\$SHOME/java/spjgraph.jar**, into the same directory.

To view each Graphlet, open the associated HTML file in a Web browser. Some of the HTML files include the Spotfire S+ code used to create the graphics. Throughout this chapter, we discuss the **fuel.html** and **map.html** Graphlets.

This Web site

(http://www.insightful.com/products/graphlets/gallery/graphlet_gallery.asp) also includes many examples using Graphlets.

CREATING A GRAPHLET DATA FILE

Spotfire S+ graphics can be saved in a Graphlet file either interactively or programmatically.

Interactive File Creation

To save a Graphlet data file interactively, create a graph in a `java.graph` window. Select the **Save As** menu item in the **File** menu. This brings up a dialog for selecting the file type and file name. Select the file type **Spotfire S+ Graphlet (SPJ)** and press **OK** to write all of the pages of the current graph into the specified file.

Programmatic File Creation

To create a Graphlet data file programmatically, call the `java.graph` function and specify a file name with the SPJ file extension. For example:

```
> java.graph(file = "myfile.spj")
```

This opens a Spotfire S+ graphics device that stores Graphlet data in the specified file. Note that this command does not open a window. Next, execute all graphics commands necessary for creating your desired graphs. When you are finished, close the device with the `dev.off` function. When the graphics device is closed, the contents are written to the named file.

Whereas some bitmap file formats such as JPEG can store only a single image in a file, the SPJ format allows multiple graph pages to be saved in the same file. When `java.graph` is used to create an SPJ file, all of the pages drawn are stored in a single file.

Page Titles and Tags

Each page in a multi-page Graphlet display has an associated *title string* and *tag string*. The title string is displayed in the tab and can be the empty string "" to show a small empty tab. If the title is the string "#", the displayed title is an auto-generated string like "Page 3". The tag string is not displayed, but is used to identify the target page in page action strings that we describe in the section [Jump to Another Page in the Graphlet](#) (page 423).

At any point during the process of creating graphics in the `java.graph` device, there is a *current page*. The title and tag for the current page are set with the following Spotfire S+ functions:

```
> args(java.set.page.title)
function(name)

> args(java.set.page.tag)
function(tag)
```

Active Regions

The `java.identify` function specifies text labels and actions to be associated with rectangular *active regions* in a graph. The Graphlet displays these labels when a user moves the mouse within the specified regions. Here is a simple example that displays the name of car models as the mouse is run over each point in a scatter plot of Weight versus Fuel.

```
# Open a java.graph device.
> java.graph(file = "fuel.spj")

# Plot the data and identify the points.
> plot(fuel.frame[["Weight"]], fuel.frame[["Fuel"]])
> java.identify(fuel.frame[["Weight"]],
+ fuel.frame[["Fuel"]],
+ labels = dimnames(fuel.frame)[[1]])

# Close the device.
> dev.off()
```

Once the file **fuel.spj** is embedded in a Web page, an end user can view this plot in a Web browser and display the name of each car model by moving the mouse pointer over each point in the graph.

We discuss the steps necessary for embedding SPJ files in the section [Embedding the Graphlet in a Web Page](#) (page 430).

The `java.identify` function takes many optional arguments, each of which we describe below.

```
> args(java.identify)
function(x1, y1, x2, y2, labels = character(0),
actions = character(0), size = 0.01, size.relative = T,
adj = 0.5, polygons = F, one.region = F)
```

x1, y1, x2, y2

These arguments specify a series of rectangles in the current plot; each rectangle defines an active region. The arguments accept numeric vectors, possibly of length one. Different combinations of the arguments can be specified as follows:

1. If all of `x1`, `y1`, `x2`, `y2` are specified, a set of arbitrary rectangular regions is defined by the points $(x1,y1)$, $(x1,y2)$, $(x2,y1)$, and $(x2,y2)$. The regions can be either inside or outside the plot region.
2. If only `x1` and `y1` are given, regions are created around each of the points $(x1,y1)$. This is how we specify the active regions in the **fuel.spj** example above.
3. If `x1` and `x2` are given, regions are created along the `x` axis from `x1` to `x2`, extending the full height of the plot region.
4. If `y1` and `y2` are given, regions are created along the `y` axis from `y1` to `y2`, extending the full width of the plot region.
5. If only `x1` is given, regions are created around these values on the `x` axis, extending the full height of the plot region.
6. If only `y1` is given, regions are created around these values on the `y` axis, extending the full width of the plot region.

The argument `x1` may also be a structure containing components `x` and `y` or a two-column matrix, in which case the two vectors are used for the `x1` and `y1` arguments, and any supplied `y1` argument is ignored.

labels

The `labels` argument is a vector of strings associated with the active regions in a plot. Each label is displayed when a user hovers the mouse over the associated region. Labels can include the new line character `"\n"` to display multiple lines of text. In the **fuel.spj** example above, we specify `labels = dimnames(fuel.frame)[[1]]`.

actions

Like `labels`, the `actions` argument is also a vector of strings associated with the active regions in a plot. It defines the actions that occur when a user left-clicks in each active region. If `length(actions)` is less than `length(labels)`, the `actions` vector is

extended with empty strings (which specify no action) to the proper length. We discuss the format of action specification strings in the section Action Strings (page 422).

size

In cases where only points are specified, the `size` argument determines how large the active regions around the points are. The `size` argument can be a vector of two numbers, which is interpreted as separate `x` and `y` values. If `size.relative=T`, the `size` values are fractions of the plot's `x` or `y` range. Otherwise, `size` is interpreted as absolute values in pixels.

size.relative

This is a logical value that determines how the `size` argument is interpreted.

adj

In cases where only points are specified, the `adj` argument determines where active regions are located relative to the points. Similar to `size`, the `adj` argument can be a vector of two numbers that specify separate `x` and `y` values. The default values position the points in the centers of the active regions. If `adj=0.0`, an active region is positioned with the associated point in the lower left corner.

polygon

If this value is `T`, this command defines one or more polygonal regions rather than rectangles. In this case, the polygon vertices are given by the `x1,y1` arguments, and `x2,y2` are ignored. The polygon vertices are listed in order, and it is assumed that the polygon closes by joining the last point to the first. Missing values (NAs) are allowed and signify breaks between polygons.

one.region

If this value is `T`, this command defines a single region composed of one or more subregions, which may be rectangles or polygons. This can be used to specify a set of non-contiguous regions that act as a single active region. If this is true, then only the first element in the `labels` and `actions` arguments is used.

Action Strings Each element of the `actions` vector for `java.identify` specifies an action to occur when the mouse is left-clicked in the associated active region. An action can be rather complicated, particularly when it pops up a menu of choices, each of which is another action. To accommodate such complicated actions in a single string, as well as to provide opportunities for future enhancements, the action string is specified in XML format. This format can be difficult to write without error, so Spotfire S+ includes the set of functions described below for creating these strings. If an action string is not in one of these formats, clicking on the active region does nothing.

Jump to Another Web Page An example XML action string for jumping to another Web page:

```
<link href="http://www.tibco.com" target="_top"/>
```

The `href` property gives a URL specifying a Web page; in this example, the Web page is **www.tibco.com**. The `target` property specifies the HTML frame where the Web page should be displayed. If `target` is not given, it defaults to `"_top"`, which replaces the current Web page shown in the Web browser. Another useful `target` value is `"_blank"`, which displays the URL in a new Web browser window. Note that some Web browsers may ignore the `target` property.

The Spotfire S+ function that corresponds to this action is `java.action.link`:

```
> args(java.action.link)
function(url, target = "_top")
```

Given a URL as a string, `java.action.link` returns an action string in the above format. This function is vectorized, so that passing in a vector of URLs returns a vector of corresponding action strings. If `target` is specified, it is used in the action string; otherwise, it defaults to `"_top"`. For example, the action string above can be generated with the following Spotfire S+ code:

```
> java.action.link("http://www.tibco.com")
[1] "<link href=\"http://www.tibco.com\"
target=\"_top\"/>"
```


Jump to Another Page in the Graphlet

An example XML action string for jumping to another page in the Graphlet:

```
<page tag="p3"/>
```

The tag property specifies the page tag to select. If none of the pages in the Graphlet have the specified tag, nothing happens when the action is selected. Page tags are defined with the `java.set.page.tag` function, as we mention in the section Page Titles and Tags (page 418).

The Spotfire S+ function that corresponds to this action is `java.action.page`:

```
> args(java.action.page)
function(tag)
```

Given a page tag, `java.action.page` returns an action string in the above format. This function is vectorized, so that passing in a vector of page tags returns a vector of corresponding action strings. For example, the action string above can be generated with the following Spotfire S+ code:

```
> java.action.page("p3")
[1] "<page tag=\"p3\"/>"
```

Pop Up a Menu of Actions

An example XML action string for popping up a menu of action choices:

```
<menu title="some actions">
  <menuitem label="go to page1">
    <page tag="p1"/>
  </menuitem>
  <menuitem label="go to URL">
    <link href="http://www.tibco.com" target="_top"/>
  </menuitem>
</menu>
```

The corresponding Spotfire S+ functions are `java.action.menu` and `java.action.menuitem`:

```
> args(java.action.menu)
function(items = character(0), title = "")
```

```
> args(java.action.menuitem)
function(action, label = "item")
```

The `java.action.menu` function takes a vector of `menuitem` actions, each of which is created by `java.action.menuitem`. It returns a string defining a single menu action command. The title for the menu is specified with the `title` argument; if `title=""`, the menu has no title. Some platforms do not support titles on pop-up menus, in which case the `title` argument is ignored.

The `java.action.menuitem` function accepts a vector of action strings and returns a vector of corresponding `menuitem` objects. Each menu item appears in the menu with the specified label. For example, the action string above can be generated with the following Spotfire S+ code:

```
> java.action.menu(
+   java.action.menuitem(
+     action = c(java.action.page("p1"),
+               java.action.link("http://www.tibco.com")),
+     label = c("go to page1", "go to URL"),
+     title = "some actions")

[1] "<menu title=\"some actions\">\n<menuitem label=\"go to
page1\"><page tag=\"p1\"/></menuitem>\n<menuitem label=\"go
to URL\"><link href=\"http://www.tibco.com\"
target=\"_top\"/></menuitem>\n</menu>"
```

Define a Selection Tag

An example XML action string for defining a selection tag:

```
<select tag="t17"/>
```

The `tag` property defines a selection tag for this active region. If this region is buttoned, all regions with this selection tag (including this region) will be selected. A user can select any number of selectable regions, see the corresponding regions selected in other graphs in the same Graphlet, and send the selection list back to a server for further processing.

The corresponding S-PLUS function:

```
java.action.select(tag)
```

Given a selection tag, this function returns an action string in the format above. This is vectorized, so passing in a vector of selection tags will return a vector of action strings.

A simple click on a selectable region selects it and deselects any other regions that might have been selected. A shift-click on a region adds it to the list of selected regions. A control-click on a region toggles the selection, selecting or deselecting the region depending on whether it was already selected.

Groups of selectable regions can be selected by sweeping out a rectangle, and then clicking the Select button. If a user sweeps a rectangular area of the graph and then clicks the Select button, all selectable regions intersecting the swept rectangle will become selected. Holding the shift key down while sweeping the rectangle causes the swept regions to be added to the selected list when the Select button is clicked. Holding the control key down while sweeping the rectangle causes the selection of the swept regions to be toggled when the Select button is clicked.

JavaScript within an HTML page can access the selected tags in a Graphlet using HTML such as the following:

```
<applet name="Graphlet1"
  code="spjgraph.class" archive="spjgraph.jar"
  width="480" height="360">
  <param name="spjgraph.filename" value="StateSelect.spj">
  <param name="spjgraph.select.button" value="on">
</applet>
<button onclick="alert(document.Graphlet1.getSelectedTags())">
  List Selected Tags
</button>
```

In this HTML code, the applet element needs a name property so that the JavaScript can refer to the Graphlet, and the parameter `spjgraph.select.button` must be specified to enable the Select button in the Graphlet. The selected tags are returned as a single string, with tags separated by commas.

XML String Utility Function

The XML format reserves certain characters for particular uses, including double quotes and the less-than sign "<". To use reserved characters in string values, you must convert them to special

sequences of characters with the S-PLUS function `java.xml.string`. This function accepts a vector of strings and converts them to strings with the XML quote sequences. For example:

```
> java.xml.string("bad characters: \"<hello & Goodbye>\"")  
  
[1] "bad characters: &quot;&lt;hello & Goodbye&gt;&quot;"
```

The `java.xml.string` function may be useful when constructing XML strings explicitly. Functions such as `java.action.link` call `java.xml.string` on all of their string arguments, so ordinarily you do not need to call this function.

A Detailed Example

In this section, we illustrate the steps necessary for creating a Graphlet data file similar to the example file **map.spj**. To view the **map.spj** Graphlet, open the file **map.html** in:

- **Windows:** `<Spotfire S+ install dir>\library\winspj`
- **UNIX:** `$$HOME/library/example5/graphlet`

See the section Using the Graphlet (page 436) for a tutorial on interacting with this Graphlet. The Graphlet we create in this section is slightly simpler than the one in **map.html**, but it captures the essence of creating SPJ files in Spotfire S+. The code we use includes some of the S-PLUS functions we have discussed so far in this chapter, including `java.set.page.title`, `java.set.page.tag`, `java.identify`, and `java.action.page`.

The **map.spj** Graphlet displays a complicated multi-page plot of census data on the racial distribution of populations in 47 U.S. cities. The data set is taken from the Geospatial & Statistical Data Center at the University of Virginia (<http://fisher.lib.virginia.edu/ccdb>) and is not part of the regular Spotfire S+ program files. To locate and label the cities on a map of the United States, we use the built-in data vectors `city.name`, `city.x`, and `city.y`.

There are three main features of the **map.spj** Graphlet:

1. The first page of the graphic displays a map of the United States. Forty-seven cities are labeled on the map, each of which corresponds to an active region. The circle that marks each city is sized according to the city's population; larger cities have larger circles.

2. Clicking on an active region in the map switches to a page that contains a bar plot of racial data for the associated city. Clicking on the page tab for a particular city also displays the bar plot.
3. Each page that contains a bar plot has an active region at the bottom that switches to the first page. This allows the user to navigate the Graphlet easily and select multiple cities to view in sequence.

Suppose the data are stored in the Spotfire S+ data set `census`. This data set has the columns listed below.

- `AreaName`: Name of the city as it appears in the built-in vector `city.name`.
- `Population`: Population of the city in 1986.
- `Pct.white`: Percentage of the 1980 population that was white.
- `Pct.black`: Percentage of the 1980 population that was black.
- `Pct.amerind`: Percentage of the 1980 population that was American Indian.
- `Pct.asiapac`: Percentage of the 1980 population that was Asian or Pacific Islander.
- `Pct.hisp`: Percentage of the 1980 population that was Hispanic.
- `x`: Equivalent to the built-in vector `city.x`, which is negative longitude of the city (in degrees) corresponding to a coordinate system set up by the `usa` function.
- `y`: Equivalent to the built-in vector `city.y`, which is latitude (in degrees) corresponding to a coordinate system set up by the `usa` function.

In the steps below, we describe the Spotfire S+ code used to create a Graphlet for these data. First, open a `java.graph` graphics device to create the file **map.spj**:

```
java.graph(file = "map.spj")
```

Next, plot a map of the U.S. and label the cities in `census`. The `symbols` function draws circles on the map, the sizes of which are determined by the cities' populations. The `text` function writes the city names next to the circles.

```

usa()
symbols(census$x, census$y,
        circles = sqrt(census$Population), inches = 0.05,
        col = 2, add = T)
text(census$x, census$y, paste("", census$AreaName),
     adj = 0, cex = 0.5, col = 3)
title("Racial distribution in US cities")

```

Label the page of the Graphlet that contains the map:

```

java.set.page.title("USA map")
java.set.page.tag("p0")

```

Next, we define active regions surrounding the points in the map. This step associates each of the cities with one of the page tags "p1", "p2", "p3", The `labels` argument to `java.identify` displays the name of the city in the upper right corner of the Graphlet when the mouse hovers over the city's active region.

```

java.identify(census$x, census$y, labels = census$AreaName,
             actions = java.action.page(
                 paste("p", 1:length(census$x), sep="")))

```

Sort the cities alphabetically and generate a bar plot for each city's racial data. Each iteration of the following for loop creates one bar plot that displays in its own Graphlet page.

```

cities <- census$AreaName
for(thecity in sort(cities))
{
  datum <- census[census$AreaName == thecity, ]
  grps <- c("Pct.white", "Pct.black", "Pct.amerind",
           "Pct.asiapac", "Pct.hisp")
  grp.eng <- c("White", "Black", "American Indian",
             "Asian/Pacific", "Hispanic")
  x <- unlist(datum[grps])
  # The barchart() command creates a new Graphlet page.
  print(barchart(grp.eng ~ x, xlim = c(0,100),
               xlab = "Percent", ylab = ""))
  title(paste("Racial distribution in", thecity))

  # Label this page of the Graphlet.
  java.set.page.title(datum$AreaName)
  java.set.page.tag(paste("p",

```

```

match(thecity, cities, nomatch = 0),
sep = "", collapse = "")

# Define an active region below the x axis in the bar plot
# that returns you to the first page in the Graphlet.
title(xlab = "[Click here to return to USA Map]",
      adj = 0.0)
java.identify(y1 = par("usr")[[3]],
              y2 = par("usr")[[3]] - 100, labels = "return to map",
              actions = java.action.page("p0"))
}

```

Finally, close the graphics device. This writes all of the information from the graphics commands to the file **map.spj**:

```
dev.off()
```

The Windows winspj Library

Because the `java.graph` device uses Java, calling the `java.graph` function starts the Java runtime system if it is not already running.

The Windows `winspj` library provides an alternative implementation of `java.graph` that does not use Java at all. You can load this library by running the following command:

```
library(winspj, first=T)
```

The library must be attached with the `first=T` argument to override the default definition of the `java.graph` function.

With this library loaded, you can create SPJ files as follows:

```
java.graph(file="myfile.spj")
```

This version of `java.graph` can create SPJ files *only*. It does not support the other file types available in `java.graph` windows.

EMBEDDING THE GRAPHLET IN A WEB PAGE

To embed a Spotfire S+ Graphlet in a Web page, several files are necessary:

1. The **spjgraph.jar** file, which contains the Graphlet Java code. This file is included in the Spotfire S+ distribution in the following locations
 - **Windows:** <Spotfire S+ install dir>\library\winspj
 - **UNIX:** \$SHOME/javaCopies of this file can be freely distributed.
2. The SPJ file containing the Spotfire S+ graphics commands to be displayed in the Graphlet. Such a file can be generated in Spotfire S+ as described in the section Creating a Graphlet Data File (page 418).
3. The HTML file defining the Web page. This can be created manually or with one of the many Web page editors available.

To embed a Spotfire S+ Graphlet as an applet within a Web page, the HTML for the Web page must include an APPLET object. For example, the following is the HTML text of a very simple Web page for the **fuel.spj** file defined in the section Active Regions (page 419):

```
<HTML>
<BODY>
A Web page with an embedded Spotfire S+ graph:<BR>
<APPLET code="spjgraph.class" ARCHIVE="spjgraph.jar"
  WIDTH="967" HEIGHT="601">
<PARAM NAME=spjgraph.filename VALUE="fuel.spj">
</APPLET>
</BODY>
</HTML>
```

Suppose this page is the file **/u/web/fuel.html**. To access the Graphlet Java classes, the file **spjgraph.jar** should be in the **/u/web** directory as well. Alternatively, **spjgraph.jar** can be in a subdirectory of **/u/web**, in which case you specify the file's location as a relative path. For example, the following APPLET command looks for **spjgraph.jar** in the directory **/u/web/graph**:


```
<APPLET code="spjgraph.class" ARCHIVE="graph/spjgraph.jar"  
  WIDTH="967" HEIGHT="601">
```

The applet parameter `spjgraph.filename` specifies the name of the SPJ graphics file to be loaded by the Graphlet. In the above example, the file `/u/web/fuel.spj` is loaded. If `spjgraph.filename` is not specified, the file **graph.spj** is loaded by default. The applet width and height are specified in the APPLET command and cannot be changed in the Web browser.

Applet Parameters

A set of optional PARAM values control both the name of the graphics file and the buttons that appear at the top of the Graphlet (see Figure 8.2). As we mention above, the parameter `spjgraph.filename` specifies the name of the SPJ graphics file to be loaded. The remaining parameters interpret any of Yes, ON, Y, or T to mean ON; anything else is interpreted as OFF. For example, adding the following elements to an APPLET command causes the mouse coordinates to display initially, and does not allow graphs to be resized by the end user.

```
<PARAM NAME=spjgraph.mouse.position VALUE="ON">  
<PARAM NAME=spjgraph.resize.buttons VALUE="OFF">
```

We briefly describe each of the available PARAM options in Table 8.1 below. For additional details on the buttons that appear in a Spotfire S+ Graphlet, see the section Using the Graphlet (page 436).

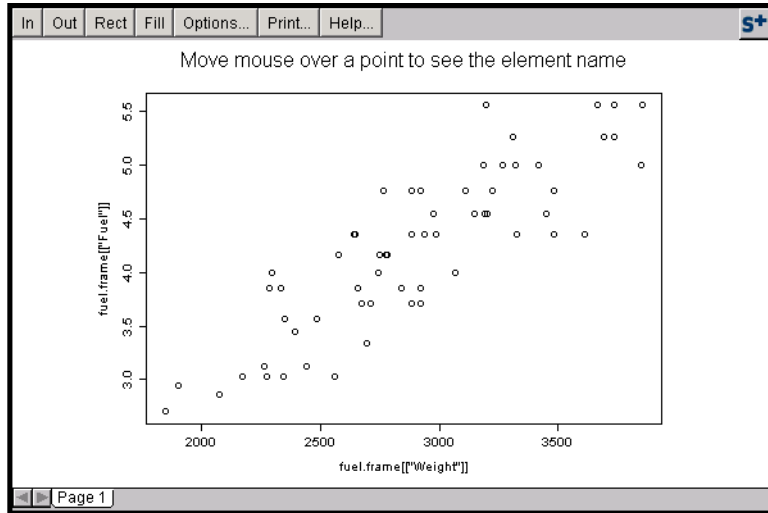


Figure 8.2: The example Graphlet in the file *fuel.html*.

Table 8.1: Applet parameters.

Applet Parameter	Description
spjgraph.filename	Specifies the name of the SPJ graphics file to be loaded by the Graphlet. Default is graph.spj .
spjgraph.mouse.position	Determines whether the mouse coordinates should be displayed when the mouse is within the graph. Default is OFF.
spjgraph.mouse.position.checkbox	Determines whether the Display mouse position check box appears in the dialog for the Options button. If it does, an end user can change whether the mouse coordinates are displayed in the Graphlet. Default is ON.

Table 8.1: *Applet parameters. (Continued)*

Applet Parameter	Description
spjgraph.active.regions	Specifies whether active regions are enabled. If they are enabled, a region is outlined and the associated label is displayed when the mouse is moved over an active region. If active regions are disabled, the regions and labels are not displayed and clicking on an active region does not perform the associated action. Default is 0N.
spjgraph.active.regions.checkbox	Determines whether the Enable active regions check box appears in the dialog for the Options button. If it does, an end user can control whether active regions are enabled in the Graphlet. Default is 0N.
spjgraph.rect.button	Determines whether the Rect button appears at the top of the Graphlet. Default is 0N.
spjgraph.resize.buttons	Determines whether any of the resize buttons (In , Out , Rect , and Fill) appear at the top of the Graphlet. If not, the graph cannot be resized by the end user. Default is 0N.
spjgraph.options.button	Determines whether the Options button appears. Default is 0N.
spjgraph.help.button	Determines whether the Help button appears. Default is 0N.

Table 8.1: *Applet parameters. (Continued)*

Applet Parameter	Description
spjgraph.tabs	Determines whether page tabs appear below the graph. Default is ON. It may be useful to turn this option off when there is only one page of graphics, or when users should switch pages only through active regions.
spjgraph.tabs.checkbox	Determines whether the Display page tabs check box appears in the dialog for the Options button. If it does, an end user can control whether page tabs appear below graphs in the Graphlet. Default is ON.
spjgraph.select.button	Determines whether the Select button appears. Default is OFF.
spjgraph.print.button	Determines whether the Print button appears. Default is ON.
spjgraph.print.option.chooser	Determines whether the Options dialog includes a list of print options: “Print current page as-is”, “Print current page unzoomed”, “Print all pages unzoomed”. Default is ON.
spjgraph.print.selections.checkbox	Determines whether the Options dialog includes the “Print selected region” check box. Default is ON.

A Detailed Example (Continued)

Here, we continue the example we began in the section *Creating a Graphlet Data File* (page 418), where we describe the steps necessary for generating the Graphlet data file **map.spj**. The HTML code below both embeds the **map.spj** file in a Web page and sets particular options in the **APPLET** command.

```
<HTML>
<HEAD>
<TITLE>Racial Distribution in US cities</TITLE>
</HEAD>
<BODY>
<APPLET code="spjgraph.class" archive="spjgraph.jar"
        WIDTH=600 HEIGHT=400 >
    <PARAM NAME=spjgraph.filename VALUE=map.spj>
    <PARAM NAME=spjgraph.mouse.position VALUE=ON>
    <PARAM NAME=spjgraph.options.button VALUE=OFF>
</APPLET>
</BODY>
</HTML>
```

Save this HTML code in a file named **map.html** and place it in a directory with the **map.spj** and **spjgraph.jar** files. After doing this, you can view and interact with the Graphlet in a Web browser. Note that the example file **map.html** in the following location is different than the HTML file we create above:

- **Windows:** <Spotfire S+ install dir> \library\winspj
- **UNIX:** \$SHOME/library/example5/graphlet

The **map.html** example file maintains the default values of all **PARAM** values, while the file we create above changes the values of **spjgraph.mouse.position** and **spjgraph.options.button**.

USING THE GRAPHLET

Suppose you are viewing a Spotfire S+ Graphlet that has been embedded in a Web page. It appears as a window with a number of labeled buttons on the top, a large region for displaying graphics, and one or more tabs along the bottom. Left-clicking on a tab displays the graphic on that page. For instance, Figure 8.3 shows the example Graphlet in the file **map.html**. This Graphlet displays the racial distribution in the populations of 47 U.S. cities. To view this Graphlet, open the **map.html** file in:

- **Windows:** <Spotfire S+ install dir> \library\winspj
- **UNIX:** \$SHOME/library/example5/graphlet

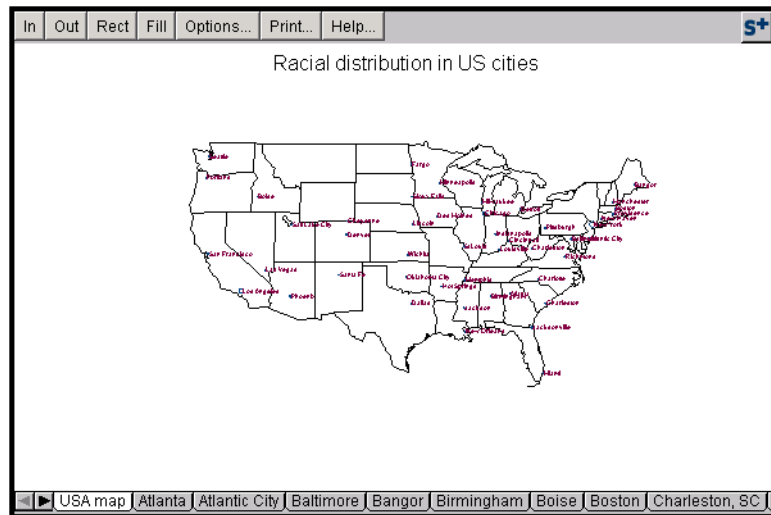


Figure 8.3: The example Graphlet in the *map.html* file.

Several buttons for resizing the image appear in a Graphlet. The **In** button zooms in on the image and the **Out** button zooms out. The **Fill** button resizes the graph so that it fills the window exactly. If the graph is resized to be larger than the window, scroll bars appear around the graph.

The **Rect** button allows zooming in on a specific region. If you press the left mouse button and drag it in the graphics window, you define the bounding box of a rectangle. After defining such a rectangle, press the **Rect** button. This changes the zoom so that the specified rectangle fills the window.

The **Select** button (not displayed by default) selects all selectable regions within a given rectangle. Click and drag the left mouse button to define a rectangle, then click the **Select** button to select all the selectable regions within the rectangle. For information about including this button in the Graphlet, see the section Applet Parameters (page 431). For more information about using this feature, see the section Define a Selection Tag (page 424).

The **Options** button displays the dialog shown in Figure 8.4. This dialog allows you to modify several options that control the operation of the Graphlet. The check box labeled **Display mouse position** specifies whether graph coordinates are displayed as the mouse runs over the graph. This box is initially cleared, so that the mouse coordinates are not displayed. A text field labeled **Mouse position digits** specifies the number of digits to use when displaying the precision of mouse coordinates. A check box labeled **Enable active regions** specifies whether active regions are enabled; this box is initially checked, so that active regions are enabled. The **Display page tabs** check box determines whether page tabs appear below the graph. By default, this box is checked and page tabs appear. It may be useful to turn this option off, however, when there is only one page of graphics.

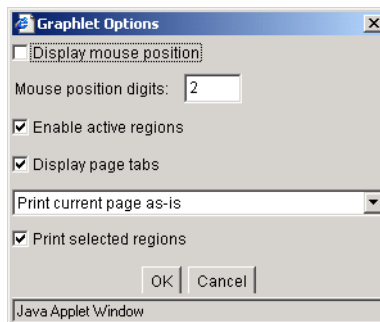


Figure 8.4: The dialog that appears when you press the **Options** button in a Spotfire S+ Graphlet.

The **Graphlet Options** dialog provides the following print controls:

- A drop-down menu with three options:
 - “Print current page as-is” (the default) specifies that the currently selected Graphlet page will be printed with the current zooming. In other words, if the current page is zoomed and panned to show a small part of the graph, only that portion will be printed.
 - “Print current page unzoomed” specifies that the entire Graphlet page will be printed with no zooming.
 - “Print all pages unzoomed” specifies that all pages of the Graphlet will be printed with no zooming.
- The “Print selected regions” check box specifies whether selected regions should be highlighted when printed.

The **Print** button prints the contents of the Graphlet without printing the rest of the Web page containing the graphlet.

The **Help** button in a Spotfire S+ Graphlet brings up a window that contains simple documentation on using the Graphlet. The **S+** button in the upper right corner displays a Web page describing the Spotfire S+ Graphlet.

Note

Not all of the buttons mentioned above may appear in a Graphlet. Applet parameters that determine which buttons appear can be set in a Web page, as described in the section Embedding the Graphlet in a Web Page (page 430). This allows a graphics creator to prevent the end user from resizing a graph, for example.

To “drill down” into the data used to create the **map.spj** Graphlet, run your mouse over the points in the map of the United States. Note that when the mouse passes over a particular city, an active region is highlighted and the name of the city appears in the upper right corner of the Graphlet. If the labels for the cities are too small on your screen, use the **Rect** button to zoom in on a particular region of the map.

If you click on one of the active regions in the map, the Graphlet jumps to another page that shows a bar plot of racial data for the associated city. For example, click on the active region for San

Francisco in the map of the United States. The Graphlet jumps to the page shown in Figure 8.5. Alternatively, you can use the arrows in the lower left corner of the Graphlet to navigate through the page tabs. Clicking on the tab titled “San Francisco” also displays the bar plot shown in the figure.

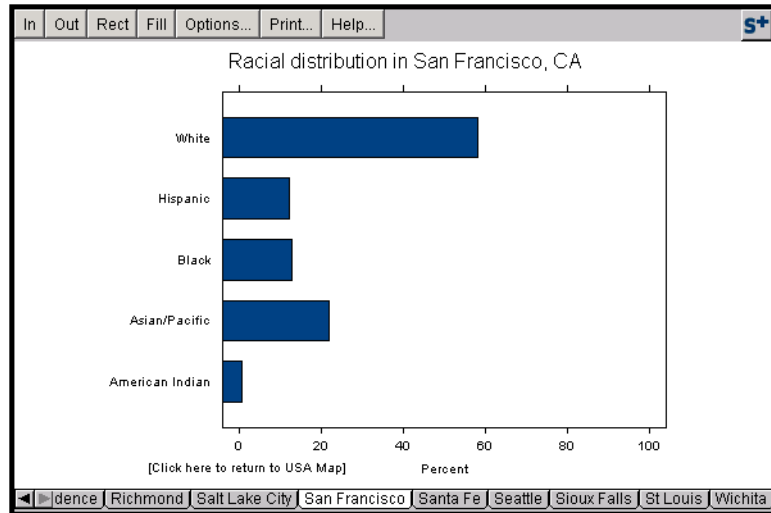


Figure 8.5: A bar plot of racial data for San Francisco in the *map.spj* Graphlet.

To return to the map of the United States, run the mouse over the lower portion of the bar plot. This highlights an active region surrounding the text “Click here to return to USA map.” Clicking within this active region returns you to the first page of the graph, where you can choose a different city to see a bar plot of its racial data.

INDEX

Numerics

- 2-D Line and Scatter Plots 324
- 2D plots
 - projecting onto a 3D plane 258
 - titles, adding axis 280
- 3D contour plots 341
- 3D line plots 339
- 3D plots
 - titles, adding axis 281
- 3D scatter plots 339

A

- abline 67, 72
- action strings 415, 422
 - jump to another page 423
 - jump to a Web page 422
 - pop up a menu of choices 423
- active regions 415, 419, 433
 - defining 420
 - defining actions 420
 - defining the location of 421
 - defining the size of 421
 - labeling 420
- add argument 98
- adding a legend 70
- adding new data to a plot 68
- adding straight lines to a scatter plot 67
- adding text to existing plot 69
- adj argument 85
- alpha channel 4
- angle argument 44

- annotation objects 309
- Annotation palette 284
- annotations 269
- aov 119
- APPLET object 430
 - spjgraph.active.regions.checkbox parameter 433
 - spjgraph.active.regions parameter 433
 - spjgraph.filename parameter 431, 432, 435
 - spjgraph.help.button parameter 433
 - spjgraph.mouse.position.checkbox parameter 432
 - spjgraph.mouse.position parameter 431, 432, 435
 - spjgraph.option.chooser parameter 434
 - spjgraph.options.button parameter 433, 434, 435
 - spjgraph.print.selections.checkbox parameter 434
 - spjgraph.rect.button parameter 433
 - spjgraph.resize.buttons parameter 431, 433
 - spjgraph.select.button parameter 434
 - spjgraph.tabs.checkbox parameter 434
 - spjgraph.tabs parameter 434
- area charts 331

- area plots 331
- arrows 102
- as.data.frame.array 195
- as.data.frame.ts 196
- aspect 191
- aspect argument 189
- at argument 87, 137
- attach 122
- auto.stats data set 72
- axes
 - formatting
 - labels 266, 277
 - titles 266
 - interactive rescaling 275
 - specifying default 2D 275
 - titles
 - adding 2D 280
 - adding 3D 281
- axes objects 309
- axes parameter 89
- axis 89
- Axis2dX object 350
- Axis2dY object 350

B

- background color 11
- backward compatibility
 - use.device.palettes function 24
 - use.legacy.graphics function 24
- bandwidth 126
- bar chart 129, 328
- Bar Chart dialog 129, 328
- barley data set 148
- barplot 43
- bar plots 328
- bar plots, 3D 340
- bg argument 17
- binomial histogram 340
- border argument 186
- box kernel 126
- box plot 133
- Box Plot dialog 133
- box plots 330
- bwplot 133

C

- candlestick plot 330
- car.miles data set 38
- cbind 32
- cex.axis argument 19
- cex.lab argument 19
- cex.main argument 19
- cex.sub argument 19
- cex argument 21, 83, 131, 169, 174
- city.name data set 104
- city.x data set 104
- city.y data set 104
- cloud 141
- cloud plot 140
- Cloud Plot dialog 140
- cm.colors color set 13
- col.axis argument 17
- col.lab argument 17
- col.main argument 17
- col.sub argument 17
- col argument 17, 21, 42, 83, 169
- color
 - alpha channel 4
 - background color 11
 - bg argument 17
 - col.axis argument 17
 - col.lab argument 17
 - col.main argument 17
 - col.sub argument 17
 - col argument 17
 - creating color sets 13
 - CSS color names 5
 - default palette 11
 - device-specific palette mode 24
 - fg argument 17
 - function summary 15
 - global palette 9
 - HSB color space 14
 - HSL color space 15
 - HSV color space 14
 - image color palette 10
 - information 6
 - legacy graphics mode 24
 - name resolution 9

- names 5
- preconstructed color sets 8
- R compatibility 2, 7, 11, 12, 13, 14, 15, 16, 18
- RGBA values 5
- RGB values 4
- setting names 7
- specification 3
- Spotfire S+ default colors 12
- Spotfire S+ default image colors 12
- transparency 4
- transparency, device support for 5
- Trellis functions 22
- truecolor 3
- X11 color names 6
- color sets
 - cm.colors 13
 - creating a color set 13
 - gray 13
 - gray.colors 13
 - grey 13
 - grey.colors 13
 - heat.colors 13
 - rainbow 13
 - terrain.colors 13
 - topo.colors 13
- color style 291
- Commands window 303
- comment plots 335
- comments 278
- composite figures 101
- conditioned Trellis graphs 358
- conditioning variables 148
- confidence bounds 289
- contour 57
- contour plot 137, 334
- contourplot 137
- Contour Plot dialog 137, 334
- coordinate systems 308
- corn.rain data set 102
- cosine kernel 126
- csi parameter 84
- CSS color names 5

- curve fit equations 269
 - editing 283
 - inserting 283
 - specifying the precision of 283
- Curve Fitting Plot dialog 283
- cuts argument 139

D

- data argument 122
- data array 63
- datax horizontal screen axis 140
- datay vertical screen axis 140
- dataz 137
- dataz perpendicular screen axis 140
- date stamps 281
- default color palette 11
- default colors 12
- default image colors 12
- defaults, saving Graph Sheet settings as 272
- density argument 44
- density plot 126, 333
 - bandwidth 126
 - cosine kernel 126
 - kernel functions 126
 - normal (Gaussian) kernel 126
 - rectangle kernel 126
 - triangle kernel 126
- density plot 127
- dev.off 118
- dev.off function 418, 429
- Device.Default 73
- digits 43
- digits argument 47
- Direct axis 93
- display properties 318
- dot plot 130
- dotplot 117, 124
- Dot Plot dialog 130
- drag-and-drop
 - adding graphs with 257
 - creating a graph with 253
 - creating a Trellis graph with 262

E

- editable graphics 322
- embedding data in Graph Sheets 293
- embedding objects 294
 - from another application 295
 - Graph Sheets 295
 - in place 296
 - updating 296
- EMF 233, 383
- equal count algorithm 157
- erase.screen 97
- error bar plots 333
- ethanol data set 169
- Export Graph dialog 298, 299
- exporting
 - graphs 298
 - with multiple pages 299
- exp parameter 91
- Extended axes label 93
- Extended Metafile 383
- Extensible Markup Language (XML)
 - see XML
- extracting data from Graph Sheets 293
- extracting panels from Trellis graphics 263
- Extract Panel/Redraw Graph 263
- eye argument 59

F

- faces 65
- fg argument 17
- fig parameter 96
- figure region 79
- FillColor property 319
- font.axis argument 19
- font.lab argument 19
- font.main argument 19
- font.sub argument 19
- font argument 169
- formula argument 120, 149, 166

- frame 96
- fuel.frame data set 143

G

- Gaussian kernel 126
- general display function 124
- general display functions 117
- glm 119
- global color palette 9
- graph area, formatting 272
- graph dialogs
 - QQ Math Plot 128
- graphics 85
 - R compatibility 2, 7, 11, 12, 13, 14, 15, 16, 18
- graphics arguments 72
- graphics dialogs
 - Bar Chart 129, 328
 - Box Plot 133
 - Cloud Plot 140
 - Contour Plot 137, 334
 - Dot Plot 130
 - Histogram 127, 333
 - Level Plot 138, 334
 - Parallel Plot 143
 - Pie Chart 131, 333
 - QQ Plot 135, 331
 - Scatter Plot Matrix 142
 - Strip Plot 134
 - Surface Plot 139, 340
 - Time Series High-Low Plot 329
- graphics objects 308
- Graphlet 436
 - action strings 415, 422
 - active regions 415, 419, 433, 438
 - and Spotfire S+ 416
 - creating 418
 - embedding in a Web page 430
 - Fill button 433, 436
 - graph coordinates of mouse 415
 - Help button 433, 438
 - In button 433, 436
 - jump to another page 423

- jump to a Web page 422
 - multiple pages 415, 418, 434
 - Options button and dialog 432, 433, 434, 437
 - Out button 433, 436
 - pop up a menu of choices 423
 - Print button 438
 - Rect button 433, 437, 438
 - resizing 415
 - Select button 437
 - setting APPLETT parameters 431
 - spjgraph.jar file 430, 435
 - Spotfire S+ button 438
 - tag string 418
 - title string 418
 - viewing and interacting with 436
 - Graph Measurements with Labels 144
 - Graph Multivariate Data 145
 - graph objects 286
 - accessing through the Object Explorer 286
 - deleting 286
 - overlapping 286
 - graphs
 - 2D, 3D, polar, matrix, and text 308
 - adding
 - using drag-and-drop 257
 - using plot buttons 256
 - adding plots to 254, 255
 - annotating 269
 - changing the plot type of 253, 254
 - creating
 - using drag-and-drop 253
 - using plot buttons 252
 - date stamps for 281
 - exporting 298
 - multiple pages 299
 - formatting 264, 272
 - adding a title to 267, 269
 - an example of 266
 - axis labels 266
 - axis titles 266
 - curve fit equations
 - adding 269, 283
 - editing 283
 - specifying the precision of 283
 - legends, adding 280, 281
 - lines, shapes, and symbols, adding 284
 - titles and subtitles, adding 280
 - labeling points on 282
 - multiline text in, adding and formatting 278
 - multipanel 263
 - multiple, on a Graph Sheet 256
 - object-oriented nature of 265
 - time stamps for 281
 - Trellis 260, 358
 - creating 262, 263
 - Graph Sheet
 - adding pages 258
 - embedded data 293
 - extracting data 293
 - formatting 271
 - placing multiple graphs on 256
 - saving defaults for 272
 - graph sheets 308
 - graph styles 291
 - gray.colors color set 13
 - gray color set 13
 - grey.colors color set 13
 - grey color set 13
 - guiDisplayDialog 321
 - guiGetArgumentNames 317
 - guiGetAxisLabelsName function 311
 - guiGetAxisName function 311
 - guiGetAxisTitleName function 311
 - guiGetGraphName function 312
 - guiGetGSName function 311
- ## H
- heat.colors color set 13

high-level graphics functions 72
high-low-open-close plot See high-low plot
high-low plot 329
hist 50
histogram 127, 333
histogram 124
Histogram dialog 127, 333
History log 303
HSB color space 14
HSL color space 15
HSV color space 14

I

identify 66
image 57
image color palette 10
internally labeled axis 93
interp 57
interpolates 57
interquartile range 133
intervals argument 161
iris data set 63

J

java.action.link function 422, 426
 target argument 422
java.action.menu function 423
 title argument 424
java.action.menuitem function 423
java.action.page function 423, 426
java.graph function 418, 419, 427
java.identify function 419, 426, 429
 actions argument 420, 422
 adj argument 421
 labels argument 420, 428
 one.region argument 421
 polygon argument 421
 size.relative argument 421
 size argument 421
 x1 argument 420
 x2argument 420
 y1 argument 420

 y2 argument 420
java.set.page.tag function 423, 426, 428
java.set.page.title function 419, 426, 428
java.set page.tag function 419
java.xml.string function 425
jitter argument 134
joint distribution 340

K

key argument 179, 183

L

labels
 axis 266
labex argument 58
lab parameter 90
legends
 adding to a graph 280, 281
level plot 138, 334
levelplot 138
Level Plot dialog 138, 334
levels 160
linear curve fit plots 327
LineColor property 319
line plots 324
 3D 339
lines 68, 174
LineStyle property 319
line types 40
LineWeight property 319
linking objects 294
 changing links 295
 editing links 295
 from another application 294
 reconnecting links 295
lm 67, 119
locator 70
loess 119
log 37
low-level graphics functions 72
low-level plotting functions 98

lty argument 21, 174
 lwd argument 21, 174

M

main argument 34
 main-effects ordering of levels 153
 main title of a plot 34
 mai parameter 80, 81
 make.groups 194
 make.symbol 106
 margin 79
 mar parameter 80, 81
 mex argument 81
 mfc col argument 79
 mfrow argument 33
 mgp argument 92
 mileage.means vector 130
 more argument 146
 most useful graphics parameters 108
 mtext 86
 multiline text, adding and
 formatting 278
 multiple plots 36
 on one graph 357

N

named colors 5
 n argument 67
 nclass argument 51
 nint argument 127
 normal (Gaussian) kernel 126

O

objects
 graph 286
 accessing through the
 Object Explorer 286
 deleting 286
 overlapping 286
 linking and embedding 294
 changing links 295
 editing links 295

from another application
 294, 295
 Graph Sheets 295
 in place 296
 updating embedded
 296
 reconnecting links 295

oma parameter 80
 omd parameter 80
 omi parameter 80
 orientation of axis labels 91
 outer margin 79
 outlier data point 66
 overlay figures 98
 ozone data set 57

P

pairs 62
 palettes
 default 11
 device specific 24
 global 9
 image color 10
 matching R 12
 overview 3
 panel 171
 panel.loess 173
 panel.special 173
 panel.superpose 179, 182
 panel.xyplot 171, 174, 175
 panel argument 193
 Panel functions 117
 panel variables 148
 par 33
 par.strip.text argument 169
 parallel 143
 parallel plot 143
 Parallel Plot dialog 143
 PARAM options 431
 spjgraph.active.regions 433
 spjgraph.active.regions.checkb
 ox 433
 spjgraph.filename 431, 432, 435
 spjgraph.help.button 433

- spjgraph.mouse.position 431, 432, 435
 - spjgraph.mouse.position.check box 432
 - spjgraph.option.chooser 434
 - spjgraph.options.button 433, 434, 435
 - spjgraph.print.selections.check box 434
 - spjgraph.rect.button 433
 - spjgraph.resize.buttons 431, 433
 - spjgraph.select.button 434
 - spjgraph.tabs 434
 - spjgraph.tabs.checkbox 434
 - p argument 176
 - pch argument 21, 41, 42, 83, 171, 174
 - pdf.graph argument 118
 - pie 45
 - pie chart 131, 333
 - piechart 132
 - Pie Chart dialog 131, 333
 - planes, inserting 259
 - plot 30
 - plot.line 177
 - plot.symbol parameter 177
 - plot area 79
 - Plot Properties dialog 288
 - plots
 - 3D line 339
 - 3D scatter 339
 - area 331
 - bar, 3D 340
 - bar charts 129, 328
 - box plots 133
 - cloud plots 140
 - comment 335
 - confidence bounds, adding to 289
 - contour plots 137, 334
 - density plots 126, 333
 - dot plots 130
 - high-low plots 329
 - histograms 127, 333
 - level plots 138, 334
 - line 324
 - lines, symbols, and colors for 289
 - parallel plots 143
 - pie charts 131, 333
 - qqplots 128, 135, 331
 - scatter 324
 - scatter plot matrix 142
 - Smith 335
 - circle 335
 - impedance 335
 - reflection 335
 - strip plots 134
 - surface plots 139, 340
 - time series plots 329
 - Trellis 263
 - Trellis graphics 148
 - type of, changing 290
 - plotting characters 41
 - plot types 37
 - points
 - identifying in a data view 282
 - labeling 282
 - points 68, 172, 174
 - polygon 174
 - position argument 146
 - precision of curve fit equations 283
 - preconstructed color sets 8
 - prepanel.loess 193
 - prepanel argument 191
 - printing
 - graphs 297
 - projection planes 341
 - pscales argument 168
 - pty argument 33
 - pugetN data set 61
- ## Q
- qqline 54
 - qqmath 128
 - QQ Math Plot dialog 128
 - qqnorm 54
 - QQ Plot dialog 135, 331
 - Q-Q plots 331

qqplots 54, 128
 normal qqplot 128
 two-dimensional 135, 331
 qqunif 55
 quantile-quantile plot See qqplots

R

rainbow color set 13
 R compatibility
 graphics 2, 7, 11, 12, 13, 14, 15,
 16, 18
 rectangle kernel See box kernel
 rectangular plot shape 33
 reorder.factor 153, 154
 rescaling axes 275
 RGBA values 5
 RGB values 4
 rotating 3D graphs 342
 Rows 183

S

scales and labels of graphs 166
 scales argument 167
 scatterplot 62
 scatter plot matrices 328
 scatter plot matrix 142
 Scatter Plot Matrix dialog 142
 scatter plots 324
 3D 339
 three-dimensional 140
 screen argument 140
 screen axes 140
 Script window 303
 segments 102, 174
 shingle 159
 show.settings 175, 177
 single-symbol operators 121
 Smith plots 335
 circle 335
 impedance 335
 reflection 335
 smooth 68
 space argument 183

span argument 174
 span parameter 187
 spjgraph.jar file 430, 435
 split argument 146
 splom 142
 square plot shape 33
 Standard axes 93
 star plot 64
 static data visualization 62
 strip.names argument 170
 strip argument 170
 strip plot 134
 stripplot 134
 Strip Plot dialog 134
 sub argument 34
 subscripts argument 174
 subset argument 122
 Subset Rows with 263
 subtitle of a plot 34
 superpose.symbol 179, 181
 surface plot 139, 340
 Surface Plot dialog 139, 340
 switzerland data set 57
 SymbolColor property 319
 symbols 104
 symbols function 428
 SymbolSize property 320
 SymbolStyle property 320

T

t 47
 tck parameter 89
 terrain.colors color set 13
 text 69, 174
 text function 428
 time series
 candlestick plots 330
 high-low plots 329
 Time Series High-Low Plot dialog
 329
 time stamps 281
 title 35, 73
 title function 428, 429
 titles

- adding to a graph 267, 269, 280
- axis 266
 - graph 267, 269
- topo.colors color set 13
- transparency 4
- transparency, device support for 5
- trellis.device 117, 175
- trellis.par.get 175
- trellis.par.set 175, 178
- Trellis graph 358
- Trellis graphics 148, 260
 - creating
 - with drag-and-drop 262
 - extracting panels 263
- Trellis settings 175
- triangle kernel 126
- truecolor 3
- type argument 31

U

- usa 104
- usa function 428
- use.device.palettes function 24
- use.legacy.graphics function 24
- using logarithmic scale 37
- usr parameter 84

W

- width argument 52, 127
- Windows Metafile 383
- wireframe 117, 124, 139
- WMF 383

X

- X11 color names 6
- xaxis argument 36
- xlab argument 35
- xlim argument 36, 167
- XML 422
 - jump to a page in a Graphlet 423
 - jump to a Web page from a Graphlet 422
 - pop-up menu in Graphlets 423
 - reserved characters 425
- xyplot 117, 120, 124

Y

- yaxis argument 36
- ylab argument 35
- ylim argument 36, 167