Computer Science and Data Analysis Series

The interface between the computer and statistical sciences is increasing, as each discipline seeks to harness the power and resources of the other. This series aims to foster the integration between the computer sciences and statistical, numerical, and probabilistic methods by publishing a broad range of reference works, textbooks, and handbooks.

Published Titles

Bayesian Artificial Intelligence
Kevin B. Korb and Ann E. Nicholson

Pattern Recognition Algorithms for Data Mining
Sankar K. Pal and Pabitra Mitra

Exploratory Data Analysis with MATLAB®
Wendy L. Martinez and Angel R. Martinez

Clustering for Data Mining: A Data Recovery Approach
Boris Mirkin

Correspondence Analysis and Data Coding with Java and R
Fionn Murtagh

R Graphics
Paul Murrell

Computer Science and Data Analysis Series

R Graphics

Paul Murrell
The University of Auckland
New Zealand

Chapman & Hall/CRC
Taylor & Francis Group
Boca Raton London New York Singapore

a more flexible basis for developing interactive plots (currently only for the Windows graphics device). This function captures key stroke events as well as mouse events and allows more general event handlers to be written as R functions.

Several add-on graphics packages provide additional interactive capabilities. The `tcltk` package provides a general facility for building GUI components and this can be used to create interactive graphics. Some of the `tcltk` demos and the `dynamicGraph` package[4] provide examples of this approach. The `Rggobi` package[33] and the `iPlots` package[62] provide an alternative approach by linking R to other graphics software applications that have sophisticated interactive features, such as brushing and linking plots[14][58].

Chapter summary

The traditional graphics system has functions to produce the standard statistical plots such as histograms, scatterplots, barplots, and piecharts. There are also functions for producing higher-dimensional plots such as 3D surfaces and contour plots and more specialized or modern plots such as dotplots, dendrograms, and mosaicplots. In most cases, the functions provide a number of arguments to allow the user to control the details of the plot, such as the widths of the boxes in a boxplot. There are a standard set of arguments for controlling the appearance of the plot (colors, fonts, line types, etc.) and the labels and axes on a plot, but these are not all available for all types of plots.

3

Customizing Traditional Graphics

Chapter preview

It is very often the case that a high-level plotting function does not produce exactly the final result that is desired. This chapter describes *low-level* traditional functions that are useful for controlling the fine details of a plot and for adding further output to a plot (e.g., adding descriptive labels).

In order to utilise these low-level functions effectively, this chapter also includes a description of the regions and coordinate systems that are used to locate the output from low-level functions. For example, there is a description of which function to use to draw text in the margins of a plot as opposed to drawing text in the data region (where the data symbols are plotted). There is also a discussion of ways to arrange several plots together on a single page.

Sometimes it is not possible to achieve a final result by modifying an existing high-level plot. In such cases, the user might need to create a plot using only low-level functions. This case is also addressed in this chapter together with some discussion of how to write a new graphics function for other people to use.

It is often the case that the default or standard output from a high-level function is not exactly what the user requires, particularly when producing graphics for publication. Various aspects of the output often need to be modified or completely replaced. This chapter describes the various ways in which the output from a traditional graphics high-level function can be customized and extended. .

The real power of the traditional graphics system lies in the ability to control many aspects of the appearance of a plot, to add extra output to a plot, and even to build a plot from scratch in order to produce precisely the right final output.

Section 3.1 introduces important concepts of drawing regions, coordinate systems, and graphics state that are required for properly working with traditional graphics at a lower level. Section 3.2 describes how to control aspects of output such as colors, fonts, line styles, and plotting symbols, and Section 3.3 addresses the problem of placing several plots on the same page. Section 3.4 describes how to customize a plot by adding extra output and Section 3.5 looks at ways to produce entirely new types of plots.

3.1 The traditional graphics model in more detail

In order to explain some of the facilities for customizing plots, it is necessary to describe more about the model underlying traditional graphics plots.

3.1.1 Plotting regions

In the base graphics system, every page is split up into three main regions: the *outer margins*, the current *figure region*, and the current *plot region*. Figure 3.1 shows these regions when there is only one figure on the page and Figure 3.2 shows the regions when there are multiple figures on the page.

The region obtained by removing the outer margins from the device is called the *inner region*. When there is only one figure, this usually corresponds to the figure region, but when there are multiple figures the inner region corresponds to the union of all figure regions.

The area outside the plot region, but inside the figure region is referred to as the *figure margins*. A typical high-level function draws data symbols and lines within the plot region and axes and labels in the figure margins or outer margins (see Section 3.4 for information on the functions used to draw output in the different regions).

The size and location of the different regions is controlled either via the `par()` function, or using special functions for arranging plots (see Section 3.3). Specifying an arrangement of the regions does not usually affect the current plot as the settings only come into effect when the next plot is started.

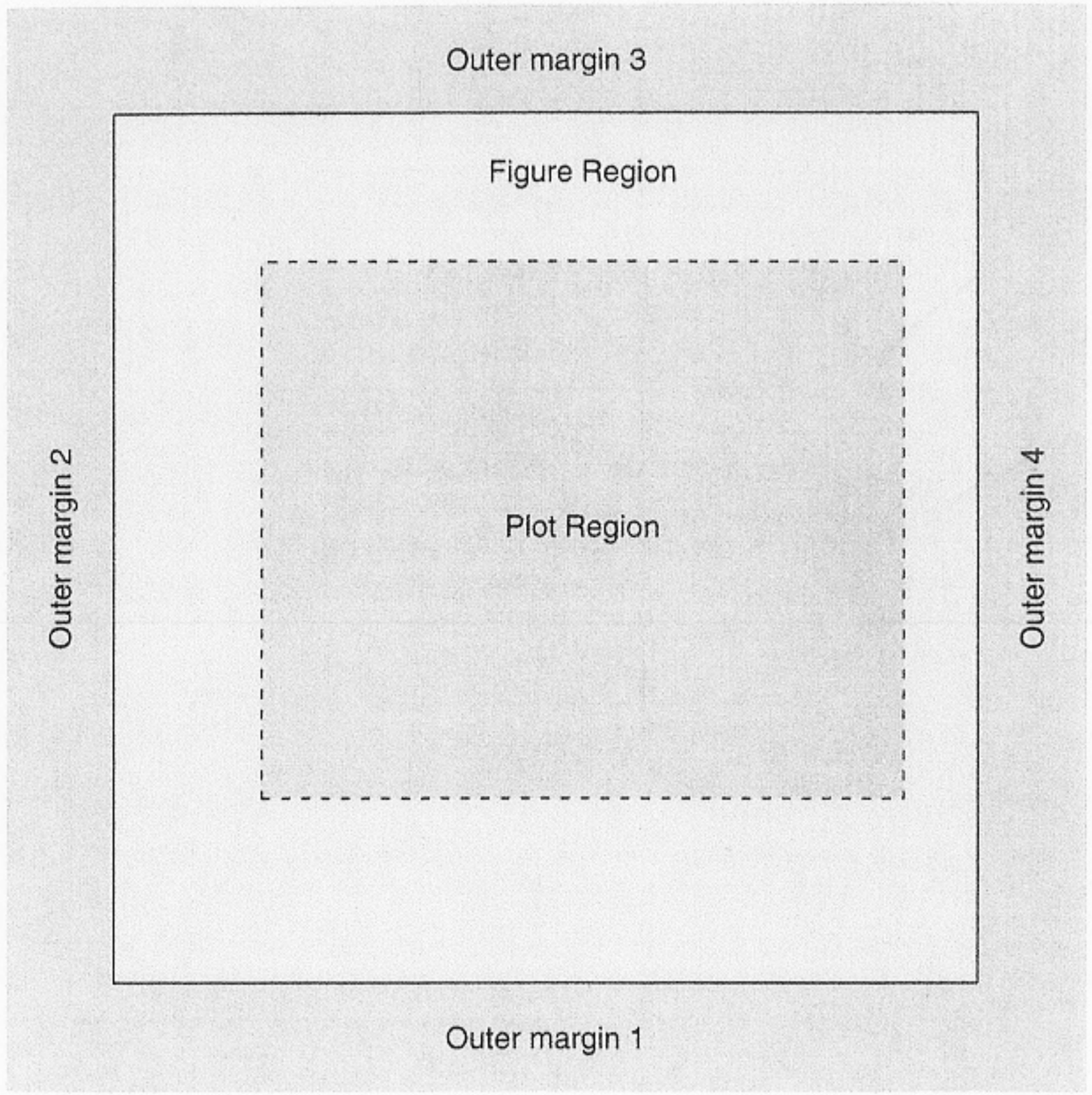


Figure 3.1
The plot regions in traditional graphics. The outer margins, figure region, and plot region, when there is a single plot on the page.

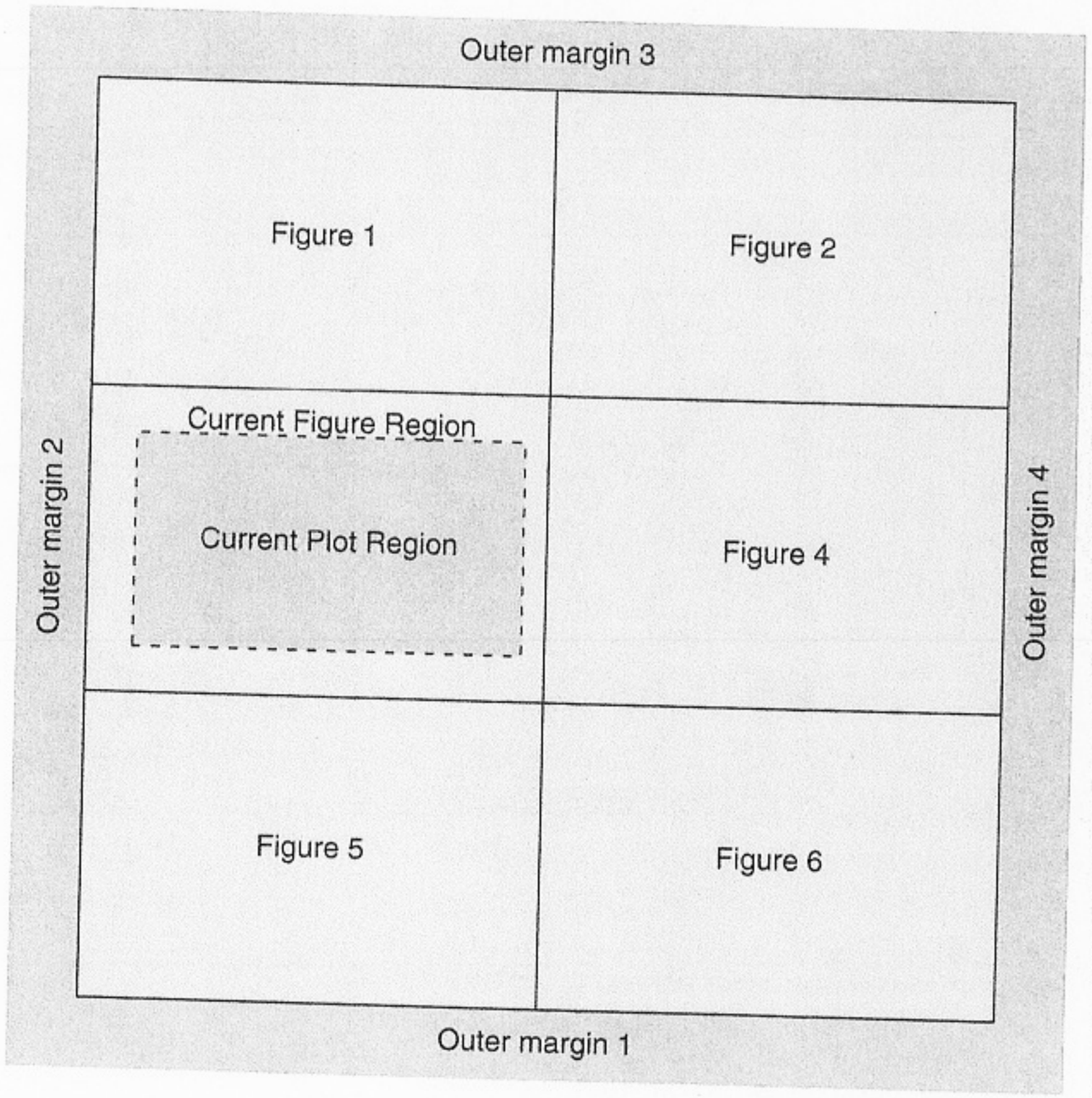


Figure 3.2
Multiple figure regions in traditional graphics. The outer margins, *current* figure region, and *current* plot region, when there are multiple plots on the page.

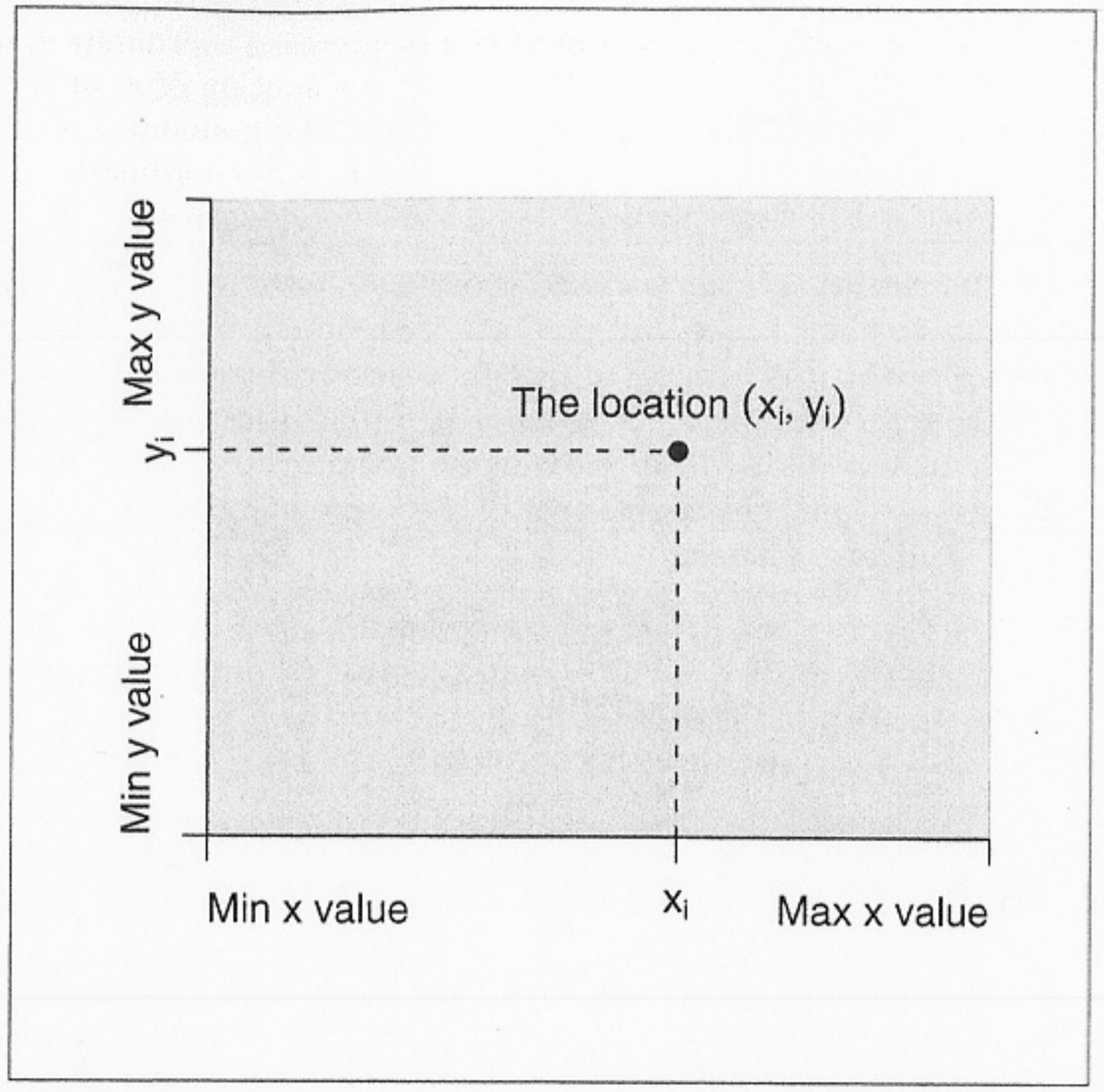


Figure 3.3
The user coordinate system in the plot region. Locations within this coordinate system are relative to the scales on the plot axes.

Coordinate systems

Each plotting region has one or more coordinate systems associated with it. Drawing in a region occurs relative to the relevant coordinate system. The coordinate system in the plot region, referred to as "user coordinates," is probably the easiest to understand as it simply corresponds to the range of values on the axes of the plot (see Figure 3.3). The drawing of data symbols, lines, text, and so on in the plot region is relative to this user coordinate system.

The scales on the axes of a plot are often set up automatically by R, but it is possible to control them explicitly via `xlim` and `ylim` arguments to high-level plotting functions (see Section 2.2.1) or via the `usr` argument to the `par()` function (see Section 3.4.7).

The figure margins contain the next most commonly-used coordinate systems. The coordinate systems in these margins are a combination of x- or y-ranges (like user coordinates) and lines of text away from the boundary of the plot region. Figure 3.4 shows two of the four figure margin coordinate systems. Axes are drawn in the figure margins using these coordinate systems.

There is a further set of "normalized" coordinate systems available for the figure margins in which the x- and y-ranges are replaced with a range from 0 to 1. In other words, it is possible to specify locations along the axes as a proportion of the total axis length. Axis labels and plot titles are drawn relative to this coordinate system. All of these figure margin coordinate systems are created implicitly from the arrangement of the figure margins and the setting of the user coordinate system.

The outer margins have similar sets of coordinate systems, but locations along the boundary of the inner region can only be specified in normalized coordinates (always relative to the extent of the complete outer margin). Figure 3.5 shows two of the four outer margin coordinate systems.

Sections 3.4.3 and 3.4.5 describe functions that produce output relative to these margin coordinate systems.

3.1.2 The traditional graphics state

The traditional graphics system maintains a graphics "state" for each graphics device. Whenever output is drawn, the graphics state is consulted to determine where it should be drawn, what color it should be, what fonts to use for text, and so on.

The graphics state consists of a large number of settings. Some of these settings describe the size and placement of the plot regions and coordinate systems described above. Some settings describe the general appearance of graphical output (the colors and line types that are used to draw lines, the fonts that are used to draw text, etc). Some settings describe aspects of the output device (e.g., the physical size of the device and the current clipping region).

Tables 3.1 to 3.3 together provide a list of all of the graphics state settings and a very brief indication of their meaning. Most of the settings are described in detail in Sections 3.2 and 3.3.

The main function used to access the graphics state is the `par()` function. Simply typing `par()` will result in a complete listing of the current graphics state. A specific state setting can be queried by supplying specific setting names as arguments to `par()`. The following code (page 52) queries the current state of the `col` and `lty` settings.

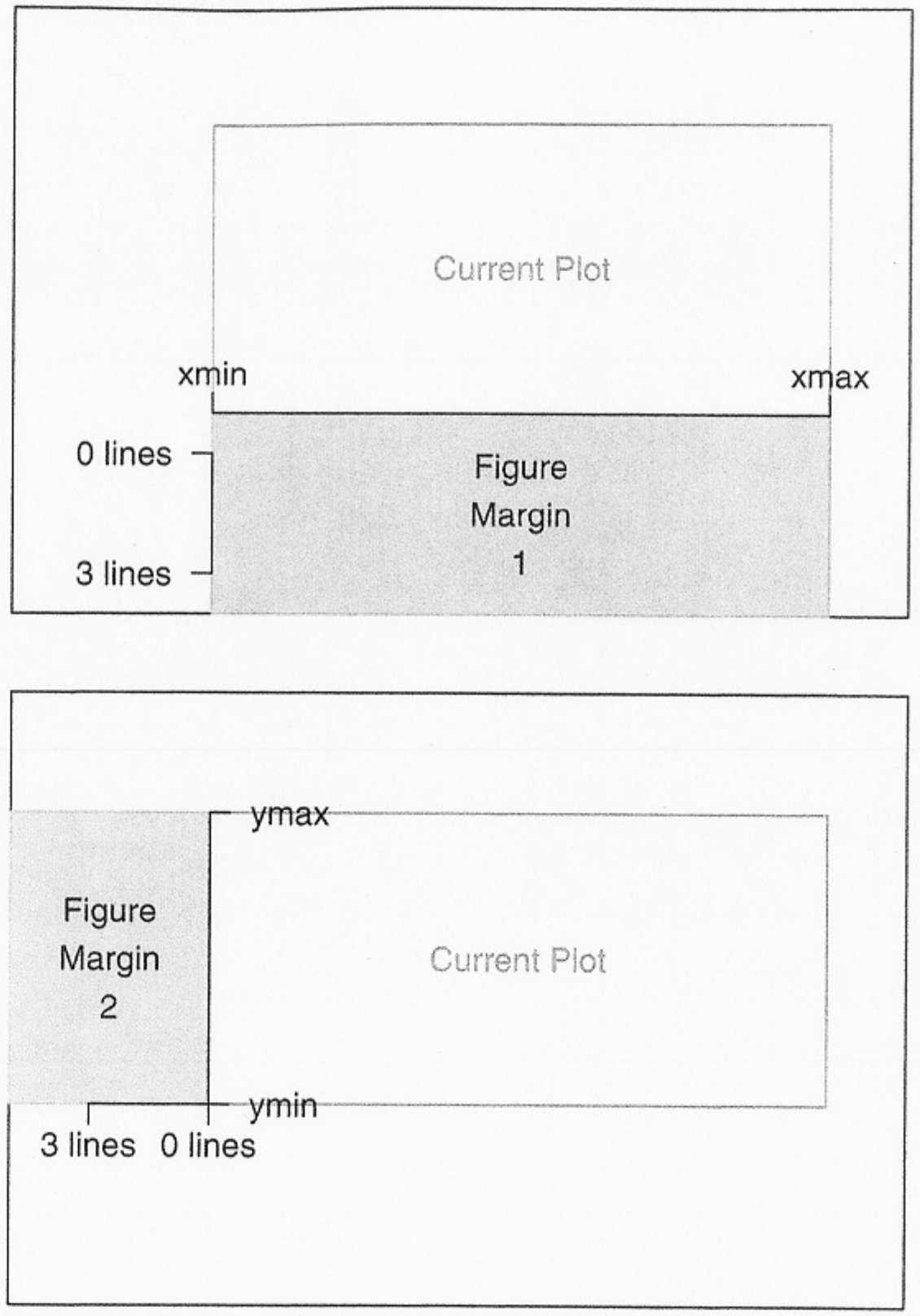


Figure 3.4
Figure margin coordinate systems. The typical coordinate systems for figure margin 1 (top plot) and figure margin 2 (bottom plot). Locations within these coordinate systems are a combination of position along the axis scale and distance away from the axis in multiples of lines of text.

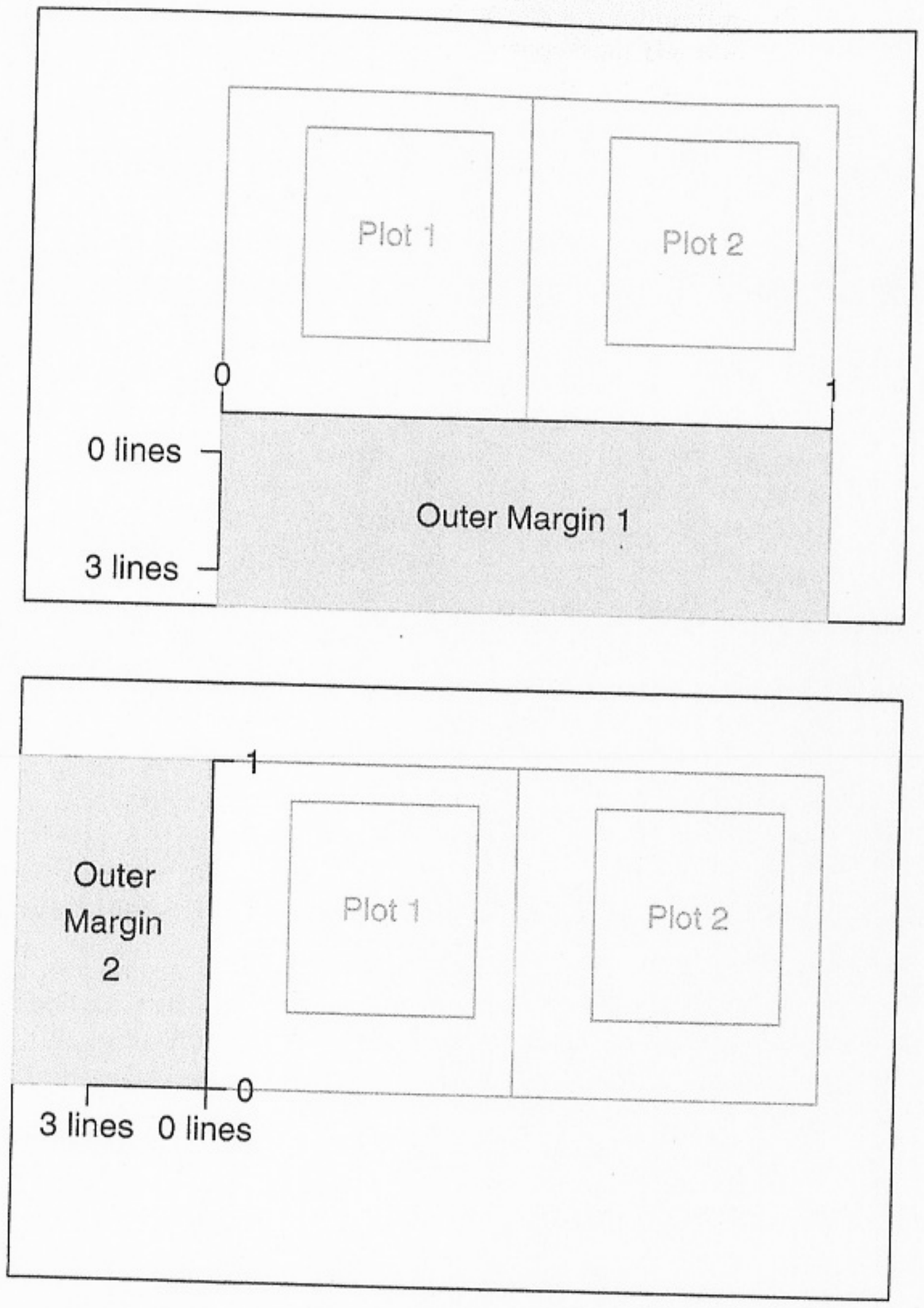


Figure 3.5
Outer margin coordinate systems. The typical coordinate systems for outer margin 1 (top plot) and outer margin 2 (bottom plot). Locations within these coordinate systems are a combination of a proportion along the inner region and distance away from the inner region in multiples of lines of text.

Table 3.1
High-level traditional graphics state settings. This set of graphics state settings can be queried and set via the `par()` function *and* can be used as arguments to other graphics functions (e.g., `plot()` or `lines()`). Each setting is described in more detail in the relevant **Section**.

Setting	Description	Section
`adj`	justification of text	3.2.3
`ann`	draw plot labels and titles?	3.2.3
`bg`	"background" color	3.2.1
`bty`	type of box drawn by `box()`	3.2.5
`cex`	size of text (multiplier)	3.2.3
`cex.axis`	size of axis tick labels	3.2.3
`cex.lab`	size of axis labels	3.2.3
`cex.main`	size of plot title	3.2.3
`cex.sub`	size of plot sub-title	3.2.3
`col`	color of lines and data symbols	3.2.1
`col.axis`	color of axis tick labels	3.2.1
`col.lab`	color of axis labels	3.2.1
`col.main`	color of plot title	3.2.1
`col.sub`	color of plot sub-title	3.2.1
`fg`	"foreground" color	3.2.1
`font`	font face (bold, italic) for text	3.2.3
`font.axis`	font face for axis tick labels	3.2.3
`font.lab`	font face for axis labels	3.2.3
`font.main`	font face for plot title	3.2.3
`font.sub`	font face for plot sub-title	3.2.3
`gamma`	gamma correction for colors	3.2.1
`lab`	number of ticks on axes	3.2.5
`las`	rotation of text in margins	3.2.3
`lty`	line type (solid, dashed)	3.2.2
`lwd`	line width	3.2.2
`mgp`	placement of axis ticks and tick labels	3.2.5
`pch`	data symbol type	3.2.4
`srt`	rotation of text in plot region	3.2.3
`tck`	length of axis ticks (relative to plot size)	3.2.5
`tcl`	length of axis ticks (relative to text size)	3.2.5
`tmag`	size of plot title (relative to other labels)	3.2.3
`type`	type of plot (points, lines, both)	3.2.4
`xaxp`	number of ticks on x-axis	3.2.5
`xaxs`	calculation of scale range on x-axis	3.2.5
`xaxt`	x-axis style (standard, none)	3.2.5
`xpd`	clipping region	3.2.7
`yaxp`	number of ticks on y-axis	3.2.5
`yaxs`	calculation of scale range on y-axis	3.2.5
`yaxt`	y-axis style (standard, none)	3.2.5

```
> par(c("col", "lty"))

$col
[1] "black"

$lty
[1] "solid"
```

The `par()` function can be used to modify traditional graphics state settings by specifying a value via an argument with the appropriate setting name. The following code sets new values for the `col` and `lty` settings.

```
> par(col="red", lty="dashed")
```

Modifying traditional graphics state settings via `par()` has a persistent effect. Settings specified in this way will hold until a different setting is specified. Settings may also be *temporarily* modified by specifying a new value in a call to a graphics function such as `plot()` or `lines()`. The following code demonstrates this idea. First of all, the line type is permanently set using `par()`, then a plot is drawn and the lines drawn between data points in this plot are dashed. Next, a plot is drawn with a temporary line type setting of `lty="solid"` and the lines in this plot are solid. When the third plot is drawn, the permanent line type setting of `lty="dashed"` is back in effect so the lines are again dashed.

```
> y <- rnorm(20)
> par(lty="dashed")
> plot(y, type="l") # line is dashed
> plot(y, type="l", lty="solid") # line is solid
> plot(y, type="l") # line is dashed
```

Only some of the graphics state settings can be set temporarily in calls to graphics functions. For example, the `mfrow` setting may not be set in this way and can only be set using `par()`. These "low level" settings are listed in Table 3.2.

A small set of graphics state settings cannot be set at all and can only be queried using `par()`. For example, there is no function to allow the user to modify the size of the current device (after the device has been created), but its size (in inches) may be obtained using `par("din")`. These "read only" settings are listed in Table 3.3.

Changes to the traditional graphics state only affect the current graphics device.

Table 3.2
Low-level traditional graphics state settings. This set of graphics state settings can be queried and set via the `par()` function. Each setting is described in more detail in the relevant **Section**.

Setting	Description	Section
`ask`	prompt user before new page?	3.2.8
`family`	font family for text	3.2.3
`fig`	location of figure region (normalized)	3.2.6
`fin`	size of figure region (inches)	3.2.6
`lend`	line end style	3.2.2
`lheight`	line spacing (multiplier)	3.2.3
`ljoin`	line join style	3.2.2
`lmitre`	line mitre limit	3.2.2
`mai`	size of figure margins (inches)	3.2.6
`mar`	size of figure margins (lines of text)	3.2.6
`mex`	line spacing in margins	3.2.6
`mfcol`	number of figures on a page	3.3.1
`mfg`	which figure is used next	3.3.1
`mfrow`	number of figures on a page	3.3.1
`new`	has a new plot been started?	3.2.8
`oma`	size of outer margins (lines of text)	3.2.6
`omd`	location of inner region (normalized)	3.2.6
`omi`	size of outer margins (inches)	3.2.6
`pin`	size of plot region (inches)	3.2.6
`plt`	location of plot region (normalized)	3.2.6
`ps`	size of text (points)	3.2.3
`pty`	aspect ratio of plot region	3.2.6
`usr`	range of scales on axes	3.4.7
`xlog`	logarithmic scale on x-axis?	3.2.5
`ylog`	logarithmic scale on y-axis?	3.2.5

Table 3.3
Read-only traditional graphics state settings. This set of graphics state settings can only be queried (via the `par()` function). Each setting is described in more detail in the relevant **Section**.

Setting	Description	Section
`cin`	size of a character (inches)	3.4.7
`cra`	size of a character ("pixels")	3.4.7
`cxy`	size of a character (user coordinates)	3.4.7
`din`	size of graphics device (inches)	3.4.7

3.2 Controlling the appearance of plots

This section is concerned with the appearance of plots, which means the colors, line types, fonts and so on that are used to draw a plot. As described in Section 3.1.2, these features are controlled via traditional graphics state settings and values are specified for the settings either with a call to the `par()` function or as arguments to a specific graphics function such as `plot()`. For example, there is a setting called `col` to control the color of output (see the next section). This can be set permanently using `par()` with an expression of the form

```
par(col="red")
```

which will affect all subsequent graphical output. Alternatively, the setting can be specified as an argument to a high-level function using an expression like

```
plot(..., col="red")
```

which means that the setting will affect the output just for that plot. Finally, the setting can be used as an argument to a low-level function, as in the expression

```
lines(..., col="red")
```

which shows that the setting can be used to control the appearance of a single piece of graphical output.

There are many individual settings that affect the appearance of a plot, but they can be grouped in terms of what aspects of a plot the settings affect. Each of the following sections details a particular group of settings, including a description of the role of individual settings and descriptions of what constitutes valid values for each setting. There are sections on: specifying colors; how to control the appearance of lines, text, data symbols, and axes; how to control the size and location of the various plotting regions; clipping (only drawing output on certain parts of the page); and specifying what should happen when a high-level function is called to start a new plot.

The appearance of plots is also affected by the location and size of the plotting regions, but this is dealt with separately in Section 3.3.

This section is not meant to be read from start to end as it is very detailed. This section should be used as a reference tool to access the relevant subsections as they are required to learn about controlling a particular aspect of a plot.

3.2.1 Colors

There are three main color settings in the traditional graphics state: `col`, `fg`, and `bg`.

The `col` setting is the most commonly used. The primary use is to specify the color of data symbols, lines, text, and so on that are drawn in the plot region. Unfortunately, when specified via a graphics function, the effect can vary. For example, a standard scatterplot produced by the `plot()` function will use `col` for coloring data symbols and lines, but the `barplot()` function will use `col` for filling the contents of its bars. In the `rect()` function, the `col` argument provides the color to fill the rectangle and there is a `border` argument specific to `rect()` that gives the color to draw the border of the rectangle. The effect of `col` on graphical output drawn in the margins also varies. It does not affect the color of axes and axis labels, but it does affect the output from the `mtext()` function. There are specific settings for affecting axes, labels, titles, and sub-titles called `col.axis`, `col.lab`, `col.main`, and `col.sub`.

The `fg` setting is primarily intended for specifying the color of axes and borders on plots. There is some overlap between this and the specific `col.axis`, `col.main`, etc. settings described above.

The `bg` setting is primarily intended to specify the color of the background for base graphics output. This color is used to fill the entire page. As with the `col` setting, when `bg` is specified in a graphics function it can have a quite different meaning. For example, the `plot()` and `points()` function use `bg` to specify the color for the interior of the data symbols, which can have different colors on the border (`pch` values `21` to `25`; see Section 3.2.4).

There is also a `gamma` setting that controls the gamma correction for a device. On most devices this can only be set once when the device is first opened.

Specifying colors

The easiest way to specify a color in R is simply to use the color's name. For example, `"red"` can be used to specify that graphical output should be (a very bright) red. R understands a fairly large set of color names (657 to be exact); type `colors()` (or `colours()`) to see a full list of known names.

It is also possible to specify colors using one of the standard color-space descriptions. For example, the `rgb()` function allows a color to be specified as

a Red-Green-Blue (RGB) triplet of intensities. Using this function, the color red is specified as `rgb(1, 0, 0)` (i.e., as much red as possible, no blue, and no green). The function `col2rgb()` can be used to see the RGB values for a particular color name.

An alternative way to provide an RGB color specification is to provide a string of the form `"#RRGGBB"`, where each of the pairs `RR`, `GG`, `BB` consist of two hexadecimal digits giving a value in the range zero (`00`) to 255 (`FF`). In this specification, the color red is given as `"#FF0000"`.

There is also an `hsv()` function for specifying a color as a Hue-Saturation-Value (HSV) triplet. The terminology of color spaces is fraught, but roughly speaking: hue corresponds to a position on the rainbow, from red (`0`), through orange, yellow, green, blue, indigo, to violet (`1`); saturation determines whether the color is dull or bright; and value determines whether the color is light or dark. The HSV specification for the (very bright) color red is `hsv(0, 1, 1)`. The function `rgb2hsv()` converts a color specification from RGB to HSV.

There is also a `convertColor()` function for converting colors between different color spaces, including the CIELAB and CIELUV color spaces[46], in which a unit distance represents a perceptually constant change in color. The `hcl()` function allows colors to be specified directly as polar coordinates within CIELUV (as a hue, chroma, and luminance triplet). This is like a perceptually uniform version of HSV.* Ross Ihaka's `colorspace` package[31] provides an alternative set of functions for generating, converting, and combining colors in a sophisticated manner in a wide variety of color spaces.

One final way to specify a color is simply as an integer index into a predefined set of colors. The predefined set of colors can be viewed and modified using the `palette()` function. In the default palette, red is specified as the integer 2.

Semitransparent colors

All R colors are stored with an alpha transparency channel. An alpha value of 0 means fully transparent and an alpha value of 1† means fully opaque. When an alpha value is not specified, the color is opaque.

The function `rgb()` can be used to specify a color with an alpha transparency channel (e.g., `rgb(1, 0, 0, 0.5)` specifies a semitransparent red), or a color can be specified as a string beginning with a `"#"` and followed by eight hexadecimal digits. In the latter case, the last two hexadecimal digits specify an alpha value in the range 0 to 255 (e.g., `"#FF000080"` specifies a semitransparent red).

A color may also be specified as `NA`, which is usually interpreted as fully transparent (i.e., nothing is drawn). The special color name `"transparent"` can also be used to specify full transparency.

Only the PDF and Quartz devices support semitransparent colors. On all other devices, semitransparent colors are rendered as fully transparent.

*The `hcl()` function is only available from R version 2.1.0.

†The maximum alpha value depends on the method being used to specify a color. When a color is specified via `rgb()`, the user can decide what the maximal value should be (it defaults to 1). When a color is specified as a string beginning with a `"#"`, the maximum value is `"FF"`.

Table 3.4
Functions to generate color sets. R functions that can be used to generate coherent sets of colors

Name	Description
`rainbow()`	Colors vary from red through orange, yellow, green, blue, and indigo, to violet.
`heat.colors()`	Colors vary from white, through orange, to red.
`terrain.colors()`	Colors vary from white, through brown, to green.
`topo.colors()`	Colors vary from white, through brown then green, to blue.
`cm.colors()`	Colors vary from light blue, through white, to light magenta.
`grey()` or `gray()`	A set of shades of grey.

Color sets

More than one color is often required within a single plot and in such cases it can be difficult to select colors that are aesthetically pleasing or are related in some way (e.g., a set of colors in which the brightness of the colors decreases in regular steps). Table 3.4 lists some functions that R provides for generating sets of colors. The output of the expression `example(rainbow)` provides a nice visual summary of the color sets generated by several of these functions.

Each of the functions in Table 3.4 selects a set of colors by taking regular steps along a path through the HSV color space. This can produce color sets that do not appear to vary smoothly. A perceptually constant color space makes it easier to generate sets of colors with even perceptual steps between

them or a set of colors that do not vary on a particular perceptual dimension. For example, the following code generates six colors from the CIELUV color space that vary regularly in terms of hue, but are all equally bright (the chroma component is fixed at 50) and all equally light (the luminance component is fixed at 60).

```
> hcl(seq(0, 360, length=7)[-7], 50, 60)

[1] "#C87A8A" "#AC8C4E" "#6B9D59" "#00A396" "#5F96C2"
[6] "#B37EBE"
```

The `RColorBrewer` package[47] provides color palettes from Cynthia Brewer's ColorBrewer tool[27]. The ColorBrewer color sets have been carefully selected by a color expert and include distinct palettes for representing nominal and ordinal categories.

The functions `colorRamp()` and `colorRampPalette()` can be used to interpolate a new color set from an existing set of colors (e.g., create additional colors from within a ColorBrewer palette).*

Device Dependency of Color Specifications

R stores colors internally as RGB triplets. The final appearance of a color can vary considerably when it is viewed on a screen, or printed on paper, or displayed through a projector as it depends on the physical characteristics of the screen, printer ink, or projector.

Fill patterns

In some cases (e.g., when printing in black and white), it is difficult to make use of different colors to distinguish between different elements of a plot. Using different levels of grey can be effective, but another option is to make use of some sort of fill pattern, such as cross-hatching. These should be used with caution because it is very easy to create visual effects that are distracting. Nevertheless, some journals actively encourage their use, so the facility has some purpose.

In R, there is only limited support for fill patterns and they can only be applied to rectangles and polygons (and only within the traditional graphics

*The functions `colorRamp()`, `colorRampPalette()`, and `convertColor()` are not available before R version 2.1.0, but some color ramp functionality is available in the `hexbin` package[10], which is part of the BioConductor project.

system). It is possible to fill a rectangle or polygon with a set of lines drawn at a certain angle, with a specific separation between the lines. A `density` argument controls the separation between the lines (in terms of lines per inch) and an `angle` argument controls the angle of the lines (in terms of degrees anti-clockwise from 3 o'clock). Examples of the use of fill patterns are given in Figures 2.4, 3.20, and their associated code.

These settings can only be controlled via arguments to the functions `rect()`, `polygon()`, `hist()`, `barplot()`, `pie()`, and `legend()` (and *not* via `par()`).

3.2.2 Lines

There are five graphics state settings for controlling the appearance of lines. The `lty` setting describes the type of line to draw (solid, dashed, dotted, ...), the `lwd` setting describes the width of lines, and the `ljoin`, `lend`, and `lmitre` settings control how the ends and corners in lines are drawn (see below).

The scope of these settings again differs depending on the graphics function being called. For example, for standard scatterplots, the setting only applies to lines drawn within the plot region. In order to affect the lines drawn as part of the axes, the `lty` setting must be passed directly to the `axis()` function.

Specifying line widths

The width of lines is specified by a simple numeric value, e.g., `lwd=3`. The interpretation of this value depends on what sort of device the line is being drawn on. In other words, the physical width of the line may be different when the line is drawn on a computer screen compared to when it is printed on a sheet of paper. On a computer screen, a line width of `1` will typically mean one pixel. For PostScript and PDF output, a line width of `1` produces a line 0.75 points wide. The default value is `1`.

Specifying line types

R graphics supports a fixed set of predefined line types, which can be specified by name, such as `"solid"` or `"dashed"`, or as an integer index (see Figure 3.6). In addition, it is possible to specify customized line types via a string of digits. In this case, each digit is a hexadecimal value that indicates a number of "units" to draw either a line or a gap. Odd digits specify line lengths and even digits specify gap lengths. For example, a dotted line is specified by `lty="13"`, which means draw a line of length one unit then a gap of length three units. A unit corresponds to the current line width, so the result scales with line width, but is device-dependent. Up to four such line-gap pairs can

be specified. Figure 3.6 shows the available predefined line types and some examples of customized line types.

Specifying line ends and joins

When drawing thick lines, it becomes important to select the style that is used to draw corners (joins) in the line and the ends of the line. R provides three styles for both cases: there is an `lend` setting to control line ends, which can be `"round"` or flat (with two variations on flat, `"square"` or `"butt"`); and there is an `ljoin` setting to control line joins, which can be `"mitre"` (pointy), `"round"`, or `"bevel"`. The differences are most easily demonstrated visually (see Figure 3.7).

When the line join style is `"mitre"`, the join style will automatically be converted to `"bevel"` if the angle at the join is too small. This is to avoid excessively pointy joins. The point at which the automatic conversion occurs is controlled by a setting called `lmitre`, which specifies the ratio of the length of the mitre divided by the line width. The default value is `10`, which means that the conversion occurs for joins where the angle is less than 11 degrees. Other standard values are `2`, which means that conversion occurs at angles less than 60 degrees, and `1.414`, which means that conversion occurs for angles less than 90 degrees. The minimum mitre limit value is `1`.

These settings can only be specified via `par()` (not as arguments to high-level or low-level graphics functions) and not all devices will respect them (especially the line mitre limit).

It is important to remember that line join styles influence the corners on rectangles and polygons as well as joins in lines.

3.2.3 Text

There are a large number of traditional graphics state settings for controlling the appearance of text. The size of text is controlled via `ps` and `cex`; the font is controlled via `font` and `family`; the justification of text is controlled via `adj`; and the angle of rotation is controlled via `srt`.

There is also an `ann` setting, which indicates whether titles and axis labels should be drawn on a plot. This is intended to apply to high-level functions, but is not guaranteed to work with all such functions (especially functions from add-on graphics packages). There are examples of the use of `ann` as an argument to high-level plotting functions in Section 3.4.

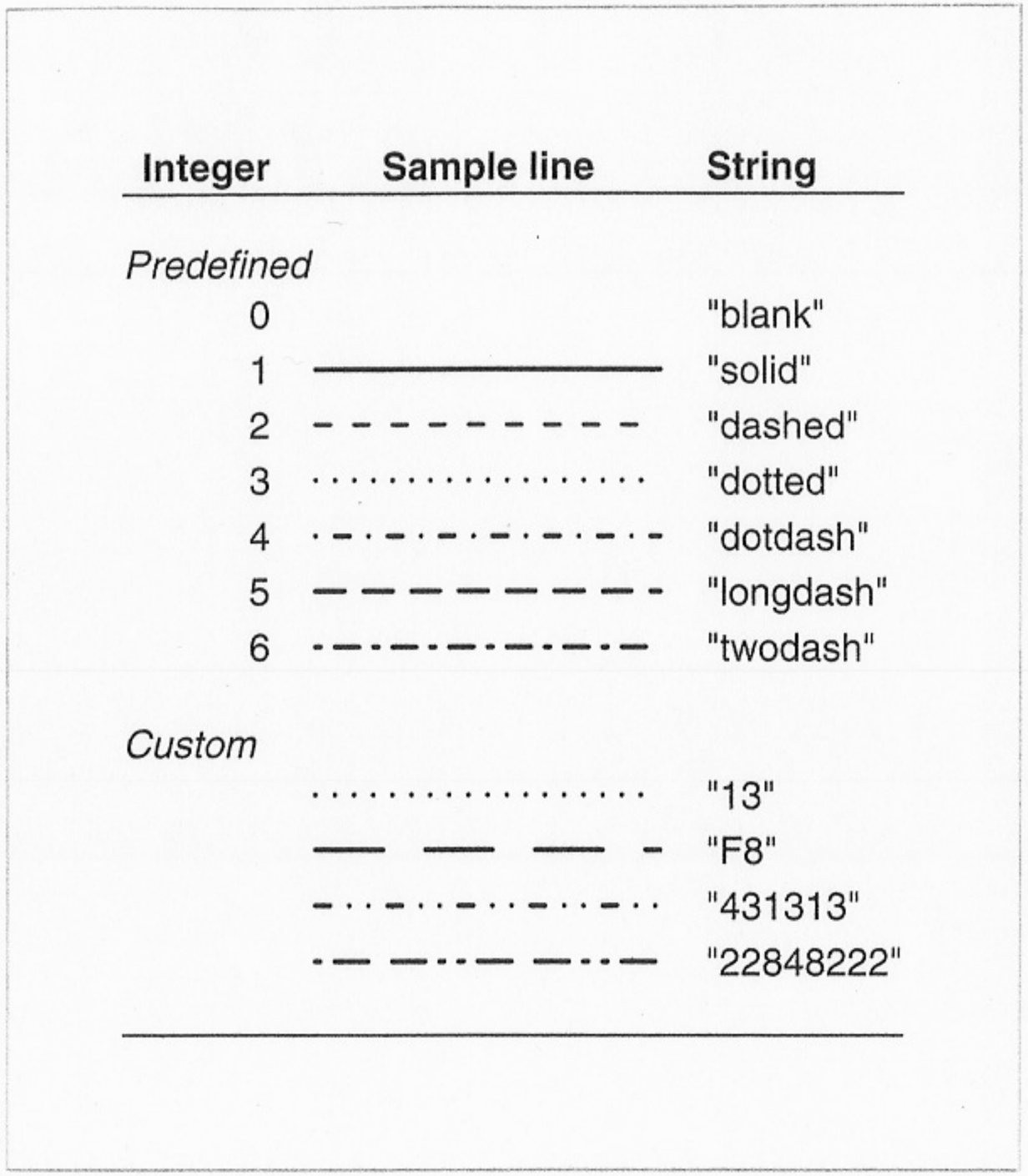


Figure 3.6
Predefined and custom line types. Line type may be specified as a predefined integer, as a predefined string name, or as a string of hexadecimal characters specifying a custom line type.

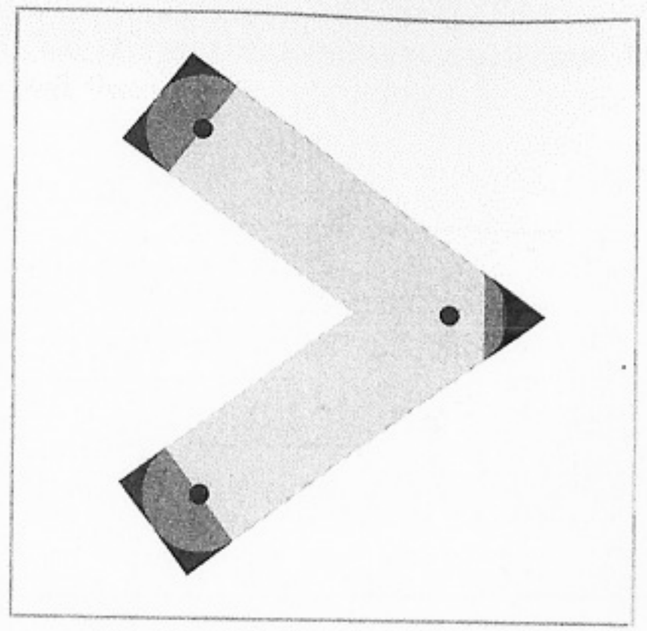


Figure 3.7
Line join and line ending styles. Three thick lines have been drawn through the same three points (indicated by black circles), but with different line end and line join styles. The black line was drawn first with `"square"` ends and `"mitre"` joins; the dark grey line was drawn on top of the black line with `"round"` ends and `"round"` joins; and the light grey line was drawn on top of that with `"butt"` ends and `"bevel"` joins.

Justification of text

The `adj` setting is a value from `0` to `1` indicating the horizontal justification of text strings (`0` means left-justified, `1` means right-justified and a value of `0.5` centers text).

The meaning of the `adj` setting depends on whether text is being drawn in the plot region, in the figure margins, or in the outer margins. In the plot region, the justification is relative to the `(x, y)` location at which the text is being drawn. In this context, it is also possible to specify two values for the setting and the second value is taken as a vertical justification for the text. Furthermore, non-finite values (`NA`, `NaN`, or `Inf`) may be specified for the justification and this is taken to mean "exact" centering. There is only a difference between a justification value of `0.5` and a non-finite justification value for vertical justification. In this case, a setting of `0.5` means text is vertically centered based on the height of the text above the text baseline (i.e., ignoring "descenders" like the tail on a "y"). A non-finite value means that text is vertically centered based on the full height of the text (including descenders). Figure 3.8 shows how various `adj` settings affect the alignment of text in the plot region.

In the figure margins and outer margins, the meaning of the `adj` setting

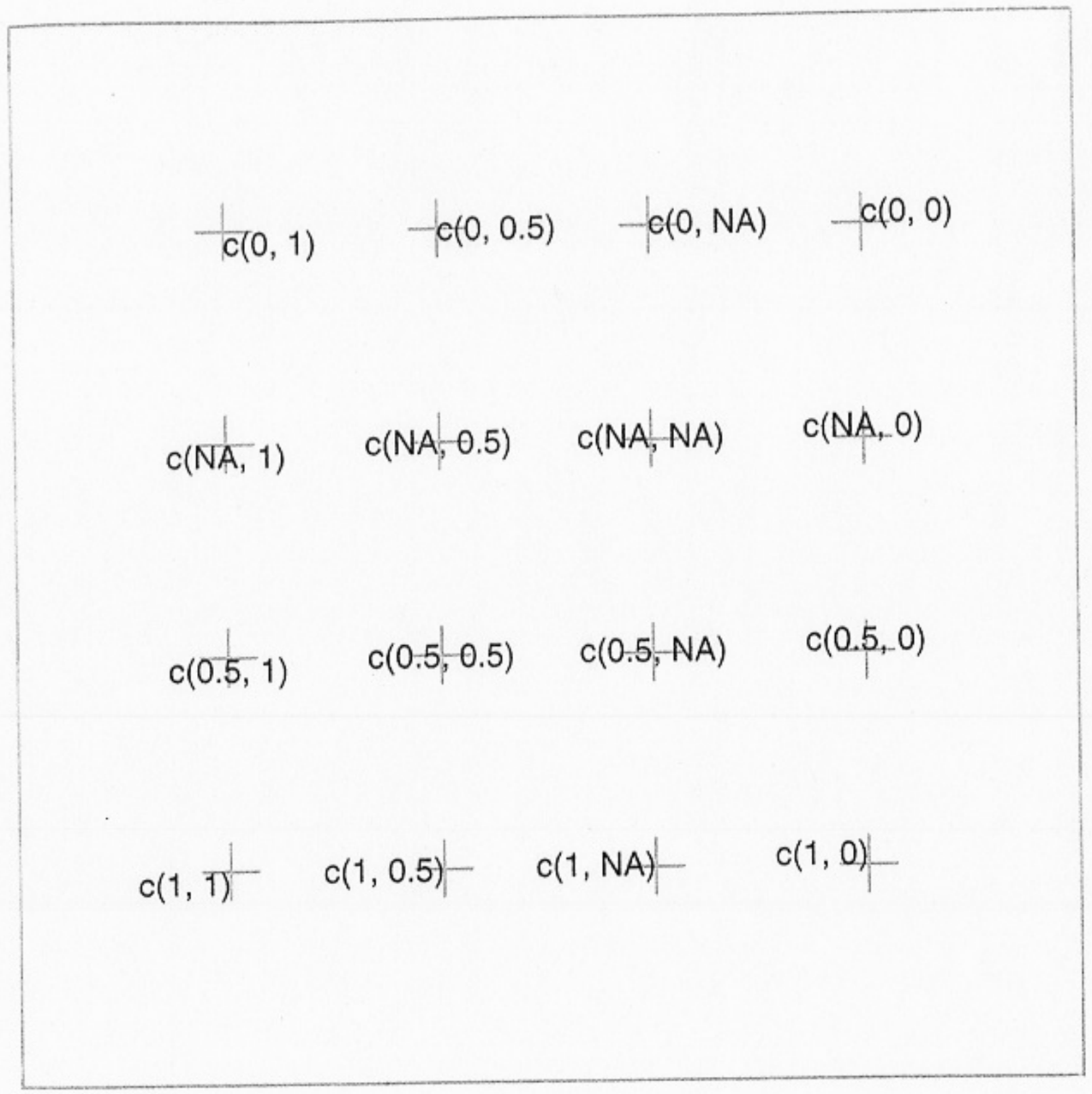


Figure 3.8
Alignment of text in the plot region. The `adj` graphical setting may be given two values, `c(`*hjust*, *vjust*`)`, where *hjust* specifies horizontal justification and *vjust* specifies vertical justification. Each piece of text in the diagram is justified relative to a grey cross to represent the effect of the relevant `adj` setting. The vertical adjustment for `NA` is subtly different from the vertical adjustment for `0.5`.

depends on the `las` setting. When margin text is parallel to the axis, `adj` specifies *both* the location and the justification of the text. For example, a value of `0` means that the text is left-justified *and* that the text is located at the left end of the margin. When text is perpendicular to the axis, the `adj` setting only affects justification. Furthermore, the `adj` setting only affects "horizontal" justification (justification in the reading direction) for text in the margins.

Rotating text

The `srt` setting specifies a rotation angle anti-clockwise from the positive x-axis, in degrees (not radians). This will only affect text drawn in the plot region (text drawn by the `text()` function). Text can be drawn at any angle within the plot region.

In the figure and outer margins, text may only be drawn at angles that are multiples of 90°, and this angle is controlled by the `las` setting. A value of `0` means text is always drawn parallel to the relevant axis (i.e., horizontal in margins 1 and 3, and vertical in margins 2 and 4). A value of `1` means text is always horizontal, `2` means text is always perpendicular to the relevant axis, and `3` means text is always vertical. This setting interacts with or overrides the `adj` and `srt` settings.

Text size

The size of text is ultimately a numerical value specifying the size of the font in "points." The font size is controlled by two settings: `ps` specifies an absolute font size setting (e.g., `ps=9`), and `cex` specifies a multiplicative modifier (e.g., `cex=1.5`). The final font size specification is simply `fontsize * cex`. On some devices, the font size that is specified will not be honored exactly. For example, when drawing in an X11 window with bitmap fonts, there are only a finite set of font sizes available and this set will vary depending on which fonts are installed. For the PostScript and PDF formats, font sizes should be accurate.

As with specifying color, the scope of a `cex` setting can vary depending on where it is given. When `cex` is specified via `par()`, it affects most text. However, when `cex` is specified via `plot()`, it only affects the size of data symbols. There are special settings for controlling the size of text that is drawn as axis tick labels (`cex.axis`), text that is drawn as axis labels (`cex.lab`), text in the title (`cex.main`), and text in the sub-title (`cex.sub`). Finally, there is a `tmag` setting for controlling the amount to magnify title text relative to other plot labels.

Multi-line text

It is possible to draw text that spans several lines, by inserting a new line escape sequence, `"\n"`, within a piece of text, as in the following example.

```
"first line\nsecond line"
```

The spacing between lines is controlled by the `lheight` setting, which is a multiplier applied to the natural height of a line of text. For example, `lheight=2` specifies double-spaced text. This setting can only be specified via `par()`.

Specifying fonts

Specifying an exact font may involve several pieces of information and is very device-specific. A font is usually part of a font "family" (e.g. Helvetica or `Courier`) and is a particular "face" within that family (e.g., **bold** or *italic*). It is also possible to specify things like the font format (e.g., TrueType or Computer Modern), the font encoding (e.g., ISO Latin 1), and even the font foundry or designer (e.g., Adobe or Sun Microsystems).

In R graphics, it is possible to specify the font face and a font family. On some devices, the latter can include extra details such as encoding.

The font face is specified via the `font` setting as an integer (Table 3.5 shows the possible values). As with color and text size, the `font` setting applies only to text drawn in the plot region. There are additional settings specifically for axes (`font.axis`), labels (`font.lab`), and titles (`font.main` and `font.sub`).

Every graphics device establishes a default font family, which is usually a sans serif font such as Helvetica or Arial. A new font family is specified via the `family` setting using a device-independent name. The names `"sans"`, `"serif"`, `"mono"`, and `"symbol"` are available for the most common devices* and provide a sans serif font, a serif font, a monospaced font, and a symbol font respectively (see Table 3.6).

Figure 3.9 demonstrates the 16 basic font family and face combinations.†

The device-independent font name is mapped to a device-dependent font family by individual devices. These mappings can be modified and new font names and mappings defined using functions such as `postscriptFont()` and `postscriptFonts()`.

*Windows, X11, Quartz, PDF, and PostScript.

†The fact that there is a font *specification* provided for all standard devices does not mean that a matching font will always be available. There can be significant differences between operating systems and locales in terms of which fonts are installed by default.

Table 3.5
Possible font face specifications in traditional graphics. The font face must be specified as an integer, usually between 1 and 4. The special value 5 indicates that a symbol font should be used. The range of valid font faces varies for different Hershey fonts, but the maximum valid value is usually 4 or less. When the font family is `"HersheySerif"`, there are a number of special font faces available.

Integer	Description
1	Roman or upright face
2	Bold face
3	Slanted or italic face
4	Bold and slanted face
5	Symbol
For the HersheySerif font family	
5	Cyrillic font
6	Slanted Cyrillic font
7	Japanese characters

Table 3.6
Device-independent and Hershey font families that are distributed with R. A font family is specified as a string

Name	Description
Device-independent fonts	
`"serif"`	Serif variable-width font
`"sans"`	Sans-serif variable-width font
`"mono"`	Mono-spaced "typewriter" font
`"symbol"`	Symbol font
Hershey fonts	
`"HersheySerif"`	Serif variable-width font
`"HersheySans"`	Sans-serif variable-width font
`"HersheyScript"`	Serif "handwriting" font
`"HersheyGothicEnglish"`	Gothic script font
`"HersheyGothicGerman"`	Gothic script font
`"HersheyGothicItalian"`	Gothic script font
`"HersheySymbol"`	Serif symbol font
`"HersheySansSymbol"`	Sans-serif symbol font

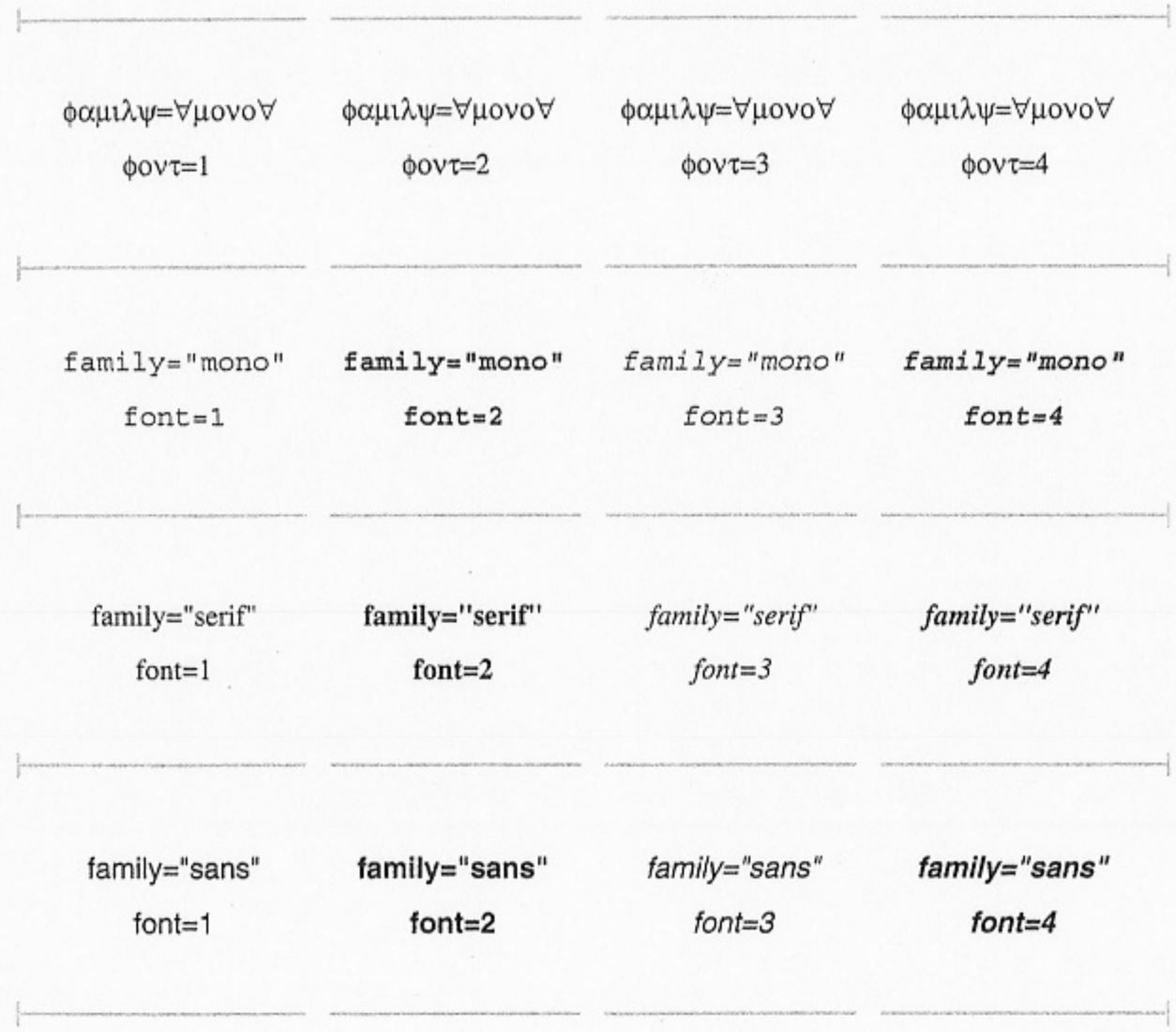


Figure 3.9
Font families and font faces. The appearance of the base sixteen font family and font face combinations that are available for X11, PDF, PostScript, Windows, and Quartz graphics devices (the output shown is for the PostScript device).

The Hershey outline fonts[1] are also distributed with R and are available for *all* output formats. The names to use with the `family` setting to obtain the different Hershey fonts are shown in Table 3.6. See the on-line help page for `Hershey` for more information on Hershey fonts.

The `family` setting can only be specified via `par()` (not as an argument to a high-level plotting function).

Locales

From R version 2.1.0, there is support for multi-byte locales, such as UTF-8 locales and East Asian locales (Chinese, Japanese, and Korean). This means that strings can be specified in R that contain characters outside of the ISO Latin 1 character set that R was restricted to prior to version 2.1.0. Such characters cannot be produced within graphical output on all devices.

As long as the appropriate fonts are available, it should be possible to produce characters outside of the ISO Latin 1 set for X11, Windows, and Quartz devices, but PostScript and PDF output can only be produced for ISO Latin 1 characters.

3.2.4 Data symbols

R provides a fixed set of 26 data symbols for plotting and the choice of data symbol is controlled by the `pch` setting. This can be an integer value to select one of the fixed set of data symbols, or a single character (see Figure 3.10). Some of the predefined data symbols (`pch` between 21 and 25) allow a fill color separate from the border color, with the `bg` setting controlling the fill color in these cases. If `pch` is a character then that letter is used as the plotting symbol. The character `"."` is treated as a special case and the device attempts to draw a very small dot (see, for example, the scatterplot matrix in Figure 2.7).

The size of the data symbols is linked to the size of text and is affected by the `cex` setting. If the data symbol is a character, the size will also be affected by the `ps` setting.

The `type` setting controls how data is represented in a plot. A value of `"p"` means that data symbols are drawn at each `(x, y)` location. The value `"l"` means that the `(x, y)` locations are connected by lines. A value of `"b"` means that both data symbols and lines are drawn. The `type` setting may also have the value `"o"`, which means that data symbols are "over-plotted" on lines (with the value `"b"`, the lines stop short of each data symbol). It is also possible to specify the value `"h"`, which means that vertical lines are drawn from the

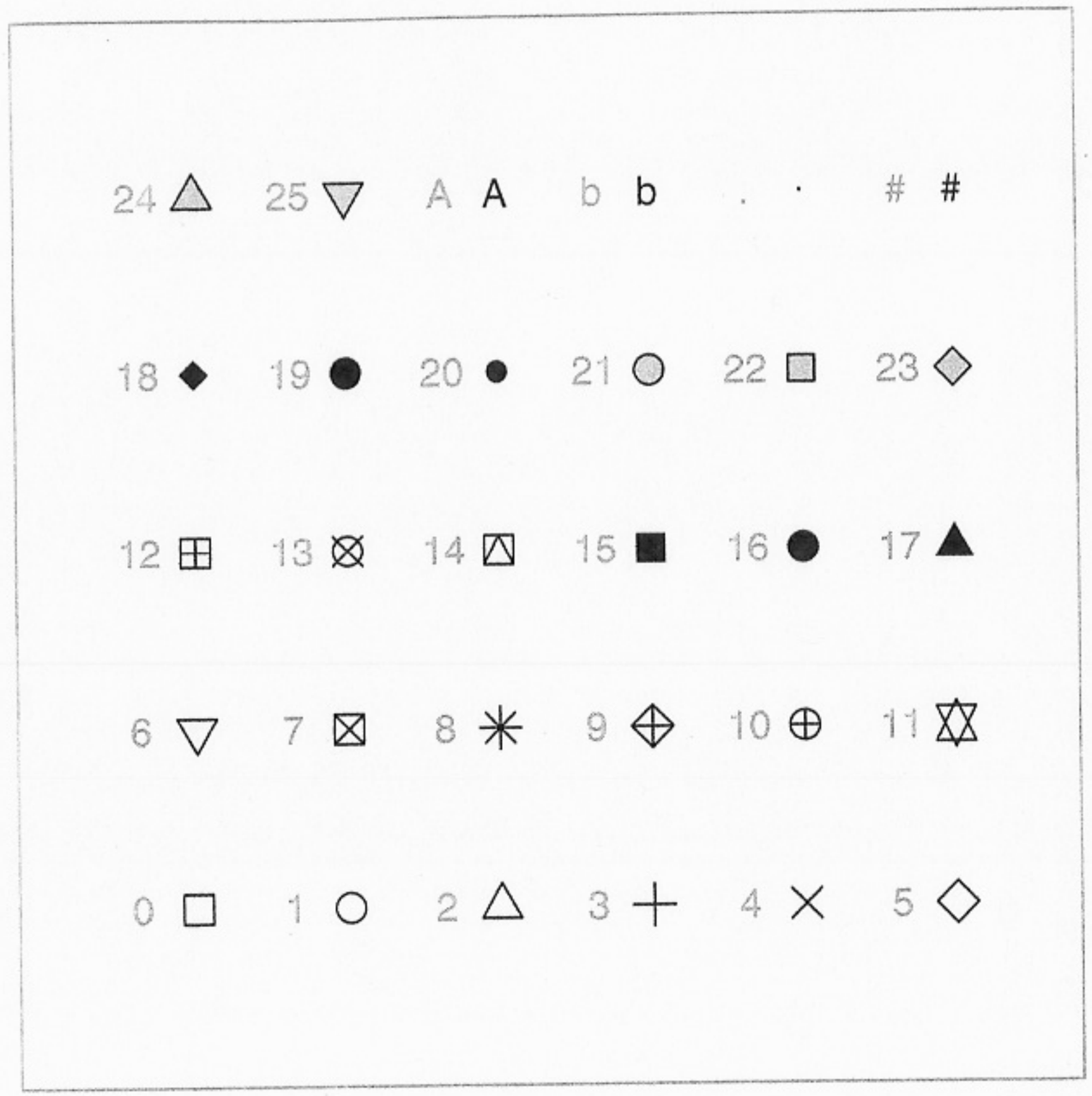


Figure 3.10
Data symbols available in R. A particular data symbol is selected by specifying an integer between 0 and 25 or a single character for the `pch` graphical setting. In the diagram, the relevant integer or character `pch` value is shown in grey to the left of the relevant symbol.

x-axis to the `(x, y)` locations (the appearance is like a barplot with very thin bars). Two further values, `"s"` and `"S"` mean that `(x, y)` locations are joined in a city-block fashion with lines going horizontally then vertically (or vertically then horizontally) between each data location. Finally, the value `"n"` means that nothing is drawn at all.

Figure 3.11 shows simple examples of the different plot types. This setting is most often specified within a call to a high-level function (e.g., `plot()`) rather than via `par()`.

3.2.5 Axes

By default, the traditional graphics system produces axes with sensible labels and tick marks at sensible locations. If the axis does not look right, there are a number of graphical state settings specifically for controlling aspects such as the number of tick marks and the positioning of labels. These are described below. If none of these gives the desired result, the user may have to resort to drawing the axis explicitly using the `axis()` function (see Section 3.4.5).

The `lab` setting in the traditional graphics state is used to control the number of tick marks on the axes. The setting is only used as a starting point for the algorithm R uses to determine sensible tick locations so the final number of tick marks that are drawn could easily differ from this specification. The setting takes two values: the first specifies the number of tick marks on the x-axis and the second specifies the number of tick marks on the y-axis.

The `xaxp` and `yaxp` settings also relate to the number and location of the tick marks on the axes of a plot. This setting is almost always calculated by R for each new plot so user settings are usually overridden (see Section 3.4.5 for an exception to this rule). In other words, it only makes sense to query this setting for its current value. The settings consist of three values: the first two specify the location of the left-most and right-most tick-marks (bottom and top tick-marks for the y-axis), and the third value specifies how many intervals there are between tick marks. When a log transformation is in effect for an axis, the three values have a different meaning altogether (see the on-line help page for `par()`).

The `mgp` setting controls the distance that the components of the axes are drawn away from the edge of the plot region. There are three values representing the positioning of the overall axis label, the tick mark labels, and the lines for the ticks. The values are in terms of lines of text away from the edges of the plot region. The default value is `c(3, 1, 0)`. Figure 3.12 gives an example of different `mgp` settings.

The `tck` and `tcl` settings control the length of tick marks. The `tcl` setting

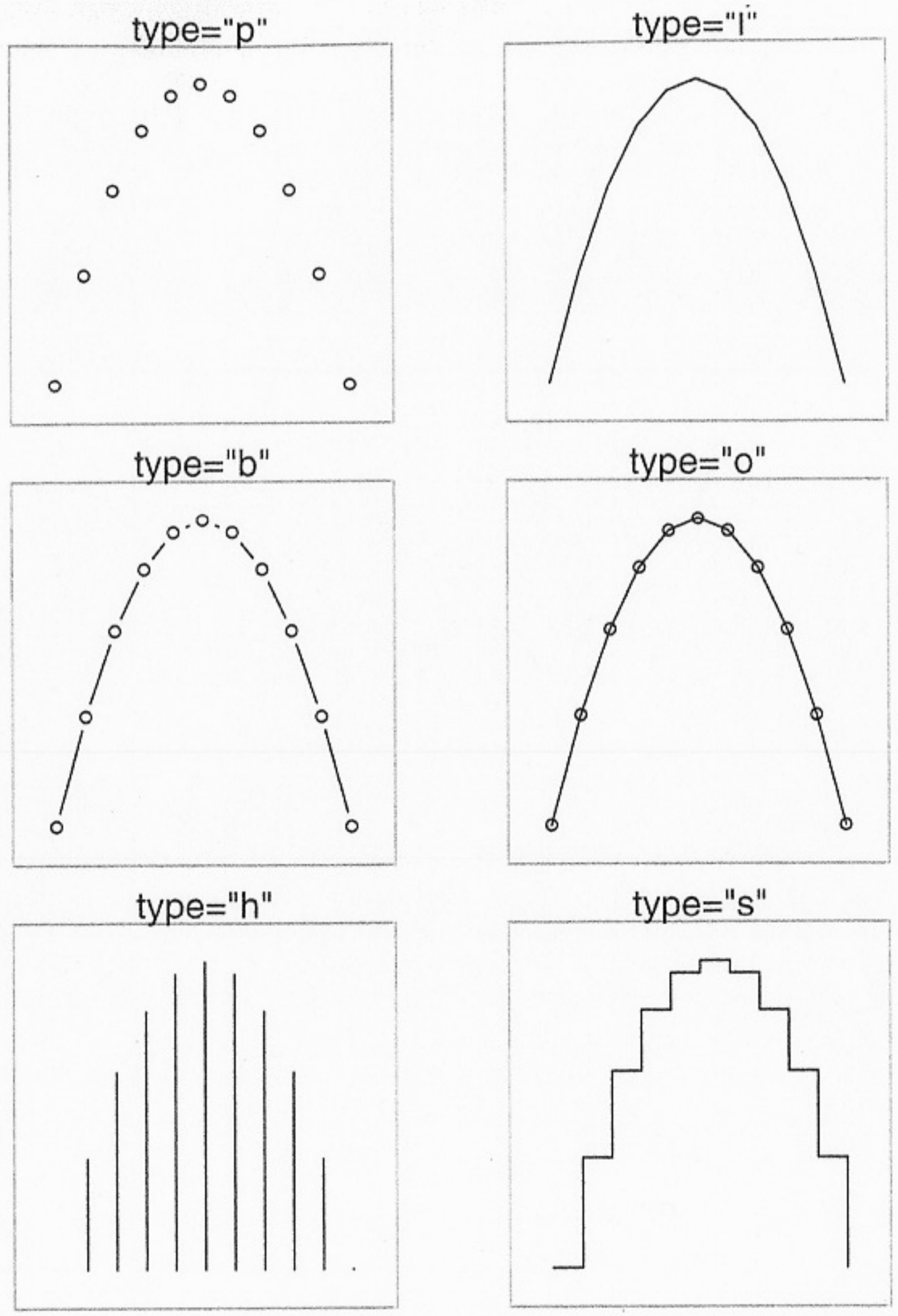


Figure 3.11
Basic plot types. Plotting the same data with different plot `type` settings. In each case, the output is produced by an expression of the form `plot(x, y, type=`*something*`)`, where the relevant value of `type` is shown above each plot.

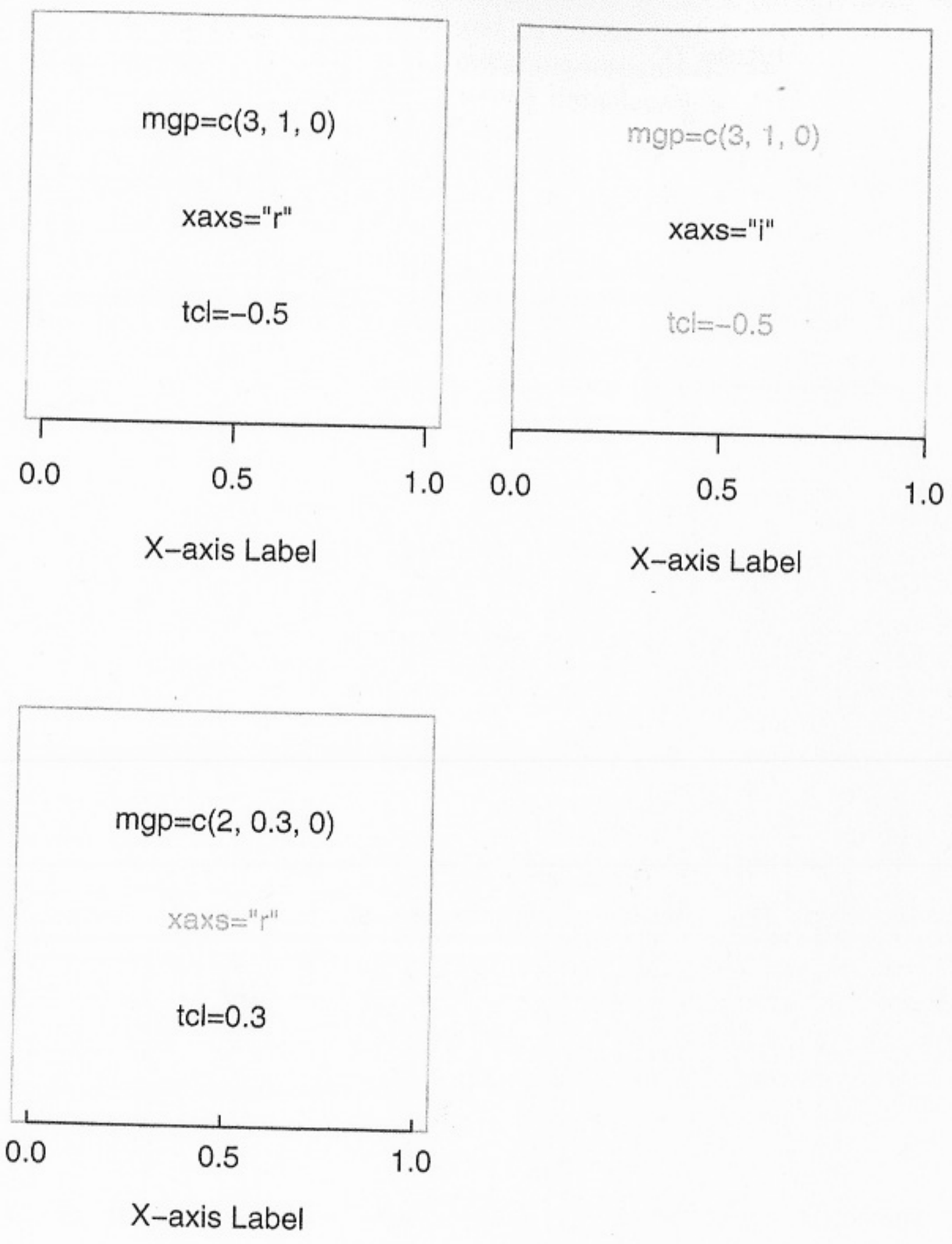


Figure 3.12
Different axis styles. The top-left plot demonstrates the default axis settings for an x-axis. The top-right plot shows the effect of specifying an "internal" axis range calculation and the bottom-left plot shows the effects of specifying different positions for the axis labels and different lengths for the tick marks.

specifies the length of tick marks as a fraction of the height of a line of text. The sign dictates the direction of the tick marks — a negative value draws tick marks outside the plot region and a positive value draws tick marks inside the plot region. The `tck` setting specifies tick mark lengths as a fraction of the smaller of the physical width or height of the plotting region, but it is only used if its value is not `NA` (and it is `NA` by default). Figure 3.12 gives an example of different `tcl` settings.

The `xaxs` and `yaxs` settings control the "style" of the axes of a plot. By default, the setting is `"r"`, which means that R calculates the range of values on the axis to be wider than the range of the data being plotted (so that data symbols do not collide with the boundaries of the plot region). It is possible to make the range of values on the axis exactly match the range of values in the data, by specifying the value `"i"`. This can be useful if the range of values on the axes are being explicitly controlled via `xlim` or `ylim` arguments to a function. Figure 3.12 gives an example of different `xaxs` settings.

The `xaxt` and `yaxt` settings control the "type" of axes. The default value, `"s"`, means that the axis is drawn. Specifying a value of `"n"` means that the axis is not drawn.

The `xlog` and `ylog` settings control the transformation of values on the axes. The default value is `FALSE`, which means that the axes are linear and values are not transformed. If this value is `TRUE` then a logarithmic transformation is applied to any values on the relevant dimension in the plot region. This also affects the calculation of tick mark locations on the axes.

When data of a special nature are being plotted (e.g., time series data), some of these settings may not apply (and may not have any sensible interpretation).

The `bty` setting is not strictly to do with axes, but it controls the output of the `box()` function, which is most commonly used in conjunction with drawing axes. This function draws a bounding box around the edges of the plot region (by default). The `bty` setting controls the type of box that the `box()` function draws. The value can be `"n"`, which means that no box is drawn, or it can be one of `"o"`, `"l"`, `"7"`, `"c"`, `"u"`, or `"]"`, which means that the box drawn resembles the corresponding uppercase character. For example, `bty="c"` means that the bottom, left, and top borders will be drawn, but the right border will not be drawn.

In addition to these graphics state settings, many high-level plotting functions, e.g., `plot()`, provide arguments `xlim` and `ylim` to control the range of the scale on the axes. Section 2.2.2 has an example.

3.2.6 Plotting regions

As described in Section 3.1.1, the traditional graphics system defines several different regions on the graphics device. This section describes how to control the size and layout of these regions using graphics state settings. Figure 3.13 shows a diagram of some of the settings that affect the widths and horizontal placement of the regions.

Outer margins

By default, there are no outer margins on a page. Outer margins can be specified using the `oma` graphics state setting. This consists of four values for the four margins in the order `(bottom, left, top, right)` and values are interpreted as lines of text (a value of 1 provides space for one line of text in the margin). The margins can also be specified in inches using `omi` or in normalized device coordinates (i.e., as a proportion of the device region) using `omd`. In the latter case, the margins are specified in the order `(left, right, bottom, top)`.

Figure regions

By default, the figure region is calculated from the settings for the outer margins, and the number of figures on the page. The figure region can be specified explicitly using either the `fig` setting or the `fin` state setting. The `fig` setting specifies the location, `(left, right, bottom, top)`, of the figure region where each value is a proportion of the "inner" region (the page less the outer margins). The `fin` setting specifies the size, `(width, height)`, of the figure region in inches and the resulting figure region is centered within the inner region.

Figure margins

The figure margins can be controlled using the `mar` state setting. This consists of four values for the four margins in the order `(bottom, left, top, right)` where each value represents a number of lines of text. The default is `c(5, 4, 4, 2) + 0.1`. The margins may also be specified in terms of inches using `mai`.

The `mex` setting controls the size of a "line" in the margins. This does not affect the size of text drawn in the margins, but is used to multiply the size of text to determine the height of one line of text in the margins.

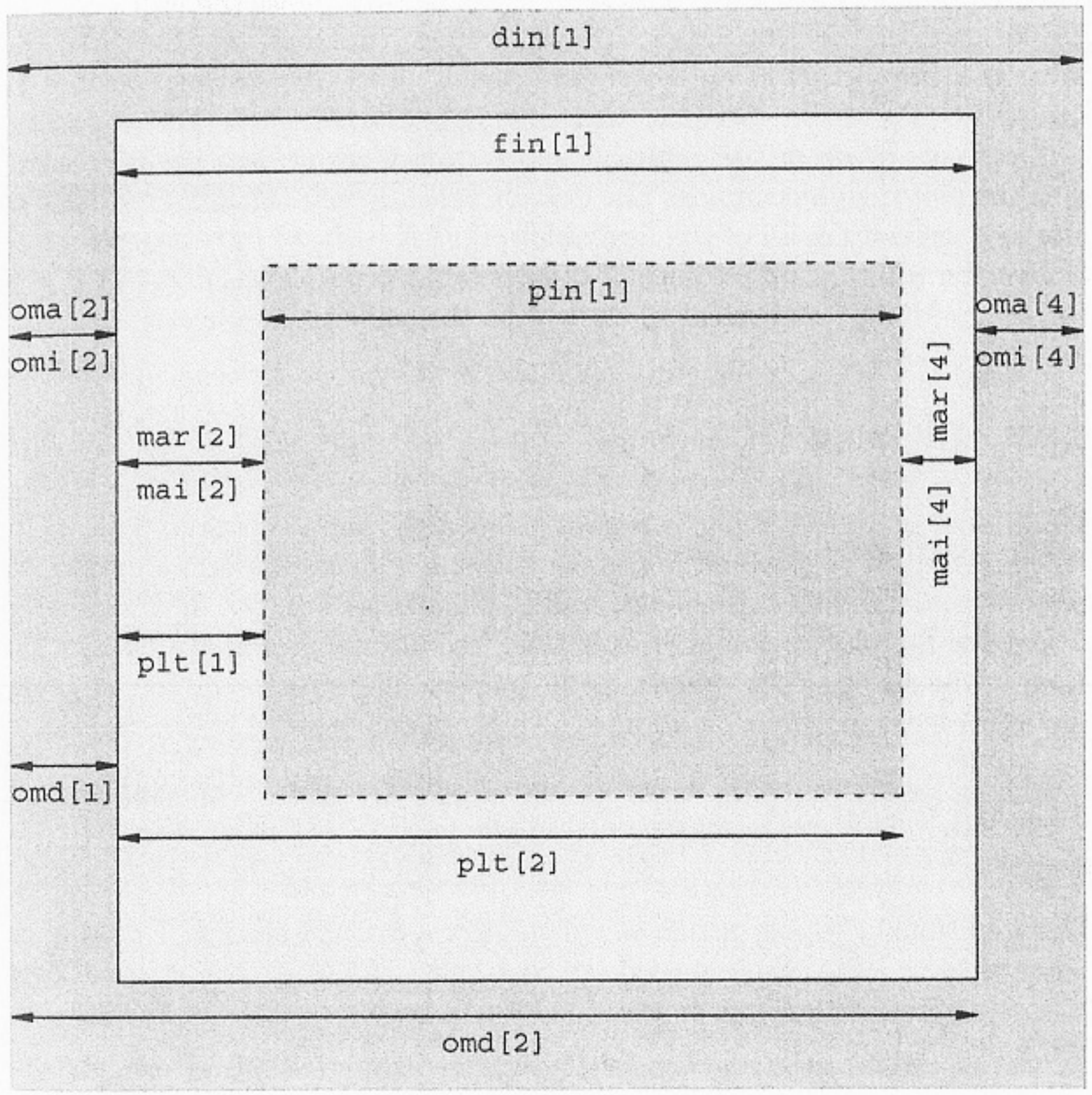


Figure 3.13
Graphics state settings controlling plot regions. These are some of the settings that control the widths and horizontal locations of the plot regions. For ease of comparison, this diagram has the same layout as Figure 3.1: the central grey rectangle represents the plot region, the lighter grey rectangle around that is the figure region, and the darker grey rectangle around that is the outer margins. A similar diagram could be produced for settings controlling heights and vertical locations.

Plot regions

By default, the plot region is calculated from the figure region less the figure margins. The location and size of the plot region may be controlled explicitly using the `plt`, `pin`, or `pty` settings. The `plt` setting allows the user to specify the location of the plot region, `(left, right, bottom, top)`, where each value is a proportion of current figure region. The `pin` setting specifies the size of the plot region, `(width, height)`, in terms of inches. The `pty` setting controls how much of the available space (figure region less figure margins) that the plot region occupies. The default value is `"m"`, which means that the plot region occupies all of the available space. A value of `"s"` means that the plot region will take up as much of the available space as possible, but it must be "square" (i.e., its physical width will be the same as its physical height).

3.2.7 Clipping

Traditional graphics output is usually clipped to the plot region. This means that any output that would appear outside the plot region is not drawn. For example, in the default behavior, data symbols for `(x, y)` locations which lie outside the plot region are not drawn. Traditional graphics functions that draw in the margins clip output to the current figure region or to the device. Section 3.4 has information about which functions draw in which regions.

It can be useful to override the default clipping region. For example, this is necessary to draw a legend outside the plot region using the `legend()` function.

The traditional clipping region is controlled via the `xpd` setting. Clipping can occur either to the whole device (an `xpd` value of `NA`), to the current figure region (a value of `TRUE`), or to the current plot region (a value of `FALSE`, which is the default).

3.2.8 Moving to a new plot

As described in Section 2.1, high-level graphics functions usually start a new plot. There are traditional graphics state settings that control exactly when and how this happens.

The `ask` setting controls whether the user is prompted before the graphics system starts a new page of output. It is useful for viewing multiple pages of output (e.g., the output from `example(boxplot)`) that otherwise flick by too fast to view properly. If the `ask` setting is `TRUE` then the user is prompted before a new page of output is begun.

The `new` setting controls whether a function that starts a new plot will move on to the next figure region (possibly a new page). Every plot sets the value to `FALSE` so that the next plot will move on by default, but if this setting has the value `TRUE` then a new plot does not move on to the next figure region. This can be used to overlay several plots on the same figure (Section 3.4.7 has an example).

3.3 Arranging multiple plots

There are a number of ways to produce multiple plots on a single page.

The number of plots on a page, and their placement on the page, can be controlled directly by specifying traditional graphics state settings using the `par()` function, or through a higher-level interface provided by the `layout()` function. The `split.screen()` function (and associated functions) provide yet another approach where a figure region can itself be treated as a complete page to split into further figure and plot regions.

These three approaches are mutually incompatible. For example, a call to the `layout()` function will override any previous `mfrow` and `mfcol` settings. Also, some high-level functions (e.g., `coplot()`) call `layout()` or `par()` themselves to create a plot arrangement, which means that the output from such functions cannot be arranged with other plots on a page.

3.3.1 Using the traditional graphics state

The number of figure regions on a page can be controlled via the `mfrow` and `mfcol` graphics state settings. Both of these consist of two values indicating a number of rows, nr, and a number of columns, nc; these settings result in $nr \times nc$ figure regions of equal size.

The top-left figure region is used first. If the setting is made via `mfrow` then the figure regions along the top row are used next from left to right, until that row is full. After that, figure regions are used in the next row down, from left to right, and so on. When all rows are full, a new page is started. For example, the following code creates six figure regions on the page, arranged in three rows and two columns and the regions are used in the order shown in Figure 3.14a.

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

If the setting is made via `mfcol`, figure regions are used by column instead of by row.

The order in which figure regions are used can be controlled by using the `mfg` setting to specify the next figure region. This setting consists of two values that indicate the row and column of the next figure to use.

3.3.2 Layouts

The `layout()` function provides an alternative to the `mfrow` and `mfcol` settings. The primary difference is that the `layout()` function allows the creation of multiple figure regions of *unequal* sizes.

The simple idea underlying the `layout()` function is that it divides the inner region of the page into a number of rows and columns, but the heights of rows and the widths of columns can be independently controlled, *and* a figure can occupy more than one row or more than one column.*

The first argument (and the only required argument) to the `layout()` function is a matrix. The number of rows and columns in the matrix determines the number of rows and columns in the layout.

The contents of the matrix are integer values that determine which rows and columns each figure will occupy. The following layout specification is identical to `par(mfrow=c(3, 2))`.

```
> layout(matrix(c(1, 2, 3, 4, 5, 6), byrow=TRUE, ncol=2))
```

It may be easier to imagine the arrangement of figure regions if the matrix is specified using `cbind()` or `rbind()`. The code below repeats the previous example, but uses `rbind()` to specify the layout matrix.

```
> layout(rbind(c(1, 2),
               c(3, 4),
               c(5, 6)))
```

The function `layout.show()` may be helpful for visualizing the figure regions that are created. The following code creates a figure visualizing the layout created in the previous example (see Figure 3.14a).

```
> layout.show(6)
```

*The underlying concept of a "layout"[43] is also implemented, in a slightly different and more general way, in the grid graphics system (see Section 5.5.6)

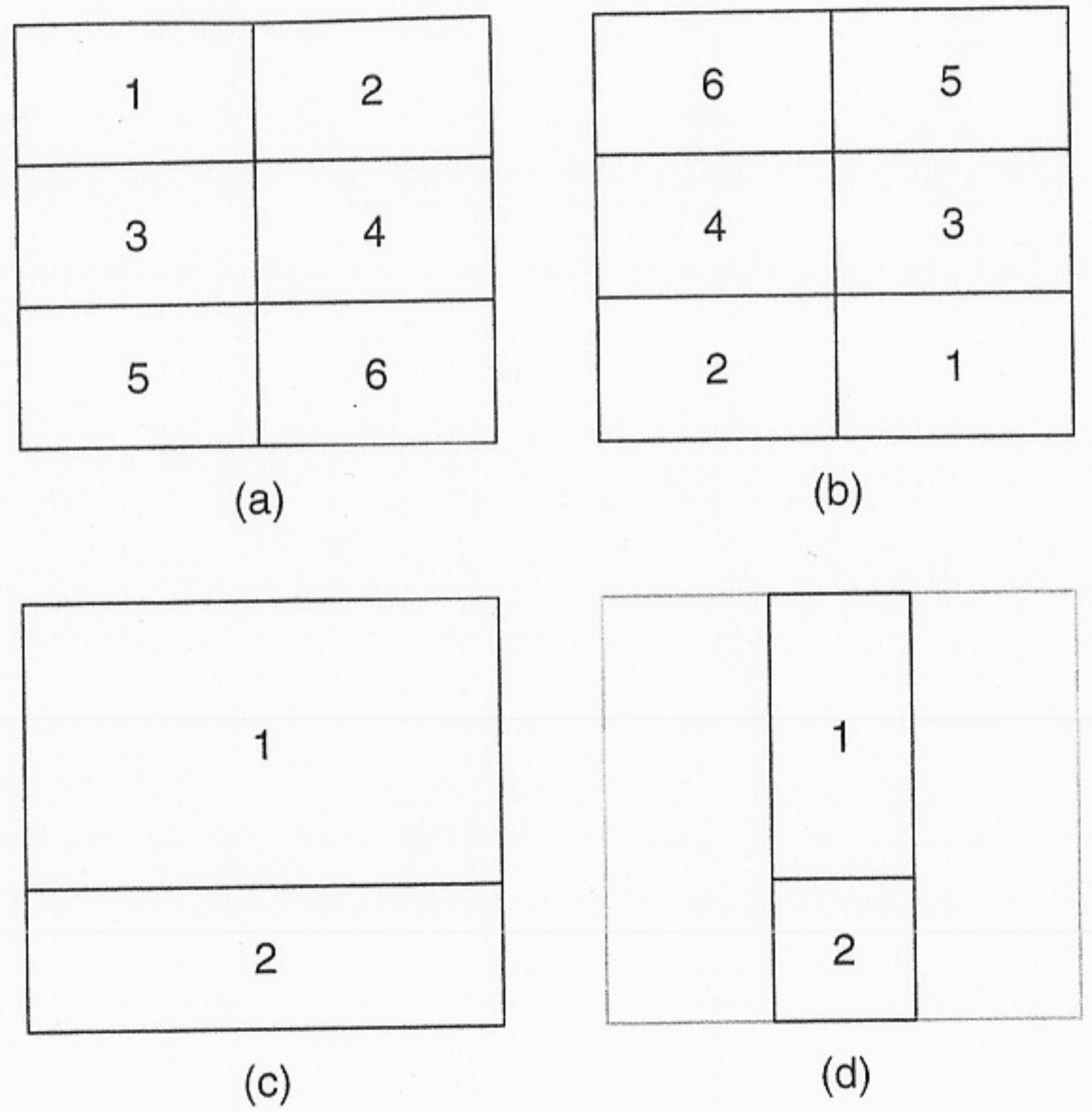


Figure 3.14
Some basic layouts. (a) A layout that is identical to `par(mfrow=c(3, 2))`. (b) Same as (a) except the figures are used in the reverse order. (c) A layout with unequal row heights. (d) same as (c) except the layout widths and heights "respect" each other.

The contents of the layout matrix determine the order in which the resulting figure regions will be used. The following code creates a layout with exactly the same rows and columns as the previous one, but the figure regions will be used in the reverse order (see Figure 3.14b).

```
> layout(rbind(c(6, 5),
               c(4, 3),
               c(2, 1)))
```

By default, all row heights are the same and all column widths are the same size and the available inner region is divided up equally. The `heights` arguments can be used to specify that certain rows are given a greater portion of the available height (for all of what follows, the `widths` argument works analogously for column widths). When the available height is divided up, the proportion of the available height given to each row is determined by dividing the row heights by the sum of the row heights. For example, in the following layout there are two rows and one column. The top row is given two-thirds of the available height $(2/(2+1))$ and the bottom row is given one third $(1/(2+1))$. Figure 3.14c shows the resulting layout.

```
> layout(matrix(c(1, 2)), heights=c(2, 1))
```

In the examples so far, the division of row heights has been completely independent of the division of column widths. The widths and heights can be forced to correspond as well so that, for example, a height of `1` corresponds to the same physical distance as a width of `1`. This allows control over the aspect ratio of the resulting figure. The `respect` argument is used to force this correspondence. The following code is the same as the previous example except that the `respect` argument is set to `TRUE` (see Figure 3.14d).

```
> layout(matrix(c(1, 2)), heights=c(2, 1),
         respect=TRUE)
```

It is also possible to specify heights of rows and widths of columns in absolute terms. The `lcm()` function can be used to specify heights and widths for a layout in terms of centimeters. The following code is the same as the previous example, except that a third, empty, region is created to provide a vertical gap of 0.5cm between the two figures (see Figure 3.15a). The `0` in the first matrix argument means that no figure will ever occupy that region.

```
> layout(matrix(c(1, 0, 2)),
         heights=c(2, lcm(0.5), 1),
         respect=TRUE)
```

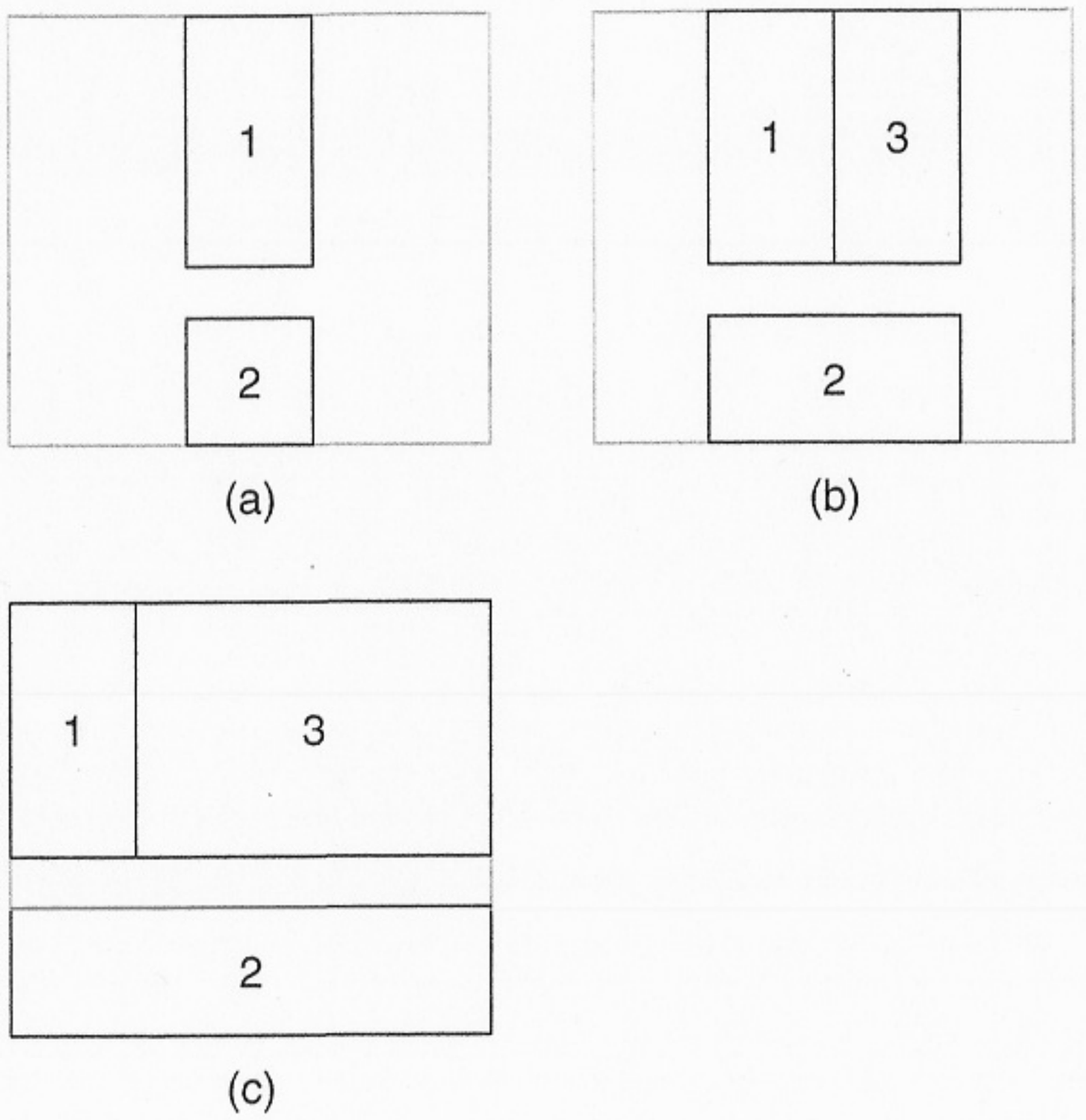



Figure 3.15
Some more complex layouts. (a) A layout with a row height specified in centimeters. (b) A layout with a figure occupying more than one column. (c) Same as (b), but with only column 1 and row 3 respected.

This next piece of code demonstrates that a figure may occupy more than one row or column in the layout. This extends the previous example by adding a second column and creating a figure region that occupies both columns of the bottom row. In the matrix argument, the value 2 appears in both columns of row 3 (see Figure 3.15b).

```
> layout(rbind(c(1, 3),
               c(0, 0),
               c(2, 2)),
         heights=c(2, lcm(0.5), 1),
         respect=TRUE)
```

Finally, it is possible to specify that only certain rows and columns should respect each other's heights/widths. This is done by specifying a matrix for the `respect` argument. In the following code, the previous example is modified by specifying that only the first column and the last row should respect each other's widths/heights. In this case, the effect is to ensure that the width of figure region 1 is the same as the height of figure region 2, but the width of figure region 3 is free to expand to the available width (see Figure 3.15c).

```
> layout(rbind(c(1, 3),
               c(0, 0),
               c(2, 2)),
         heights=c(2, lcm(0.5), 1),
         respect=rbind(c(0, 0),
                       c(0, 0),
                       c(1, 0)))
```

3.3.3 The split-screen approach

The `split.screen()` function provides yet another way to divide the page into a number of figure regions. The first argument, `figs`, is either two values specifying a number of rows and columns of figures (i.e., like the `mfrow` setting), or a matrix containing a figure region location, `(left, right, bottom, top)`, on each row (i.e., like a `fig` setting on each row).

Having established figure regions in this manner, a figure region is used by calling the `screen()` function to select a region. This means that the order in which figures are used is completely under the user's control, and it is possible to reuse a figure region, though there are dangers in doing so (the on-line help for `split.screen()` discusses this some more). The function `erase.screen()` can be used to clear a defined screen and `close.screen()` can be used to remove one or more screen definitions.

An even more useful feature of this approach is that each figure region can itself be divided up by a further call to `split.screen()`. This allows complex arrangements of plots to be created.

The downside to this approach is that it does not fit very nicely with the underlying traditional graphics system model (see Section 3.1). The recommended way to achieve complex arrangements of plots is via the `layout()` function (see Section 3.3.2) or by using the grid graphics system (see Part II), possibly in combination with traditional high-level functions (see Appendix B).

3.4 Annotating plots

Sometimes it is not enough to be able to modify the default output from high-level functions and in many situations, further graphical output must be added to achieve the desired result (see, for example, Figure 1.3). R graphics in general is fundamentally oriented to supporting the annotation of plots — the ability to add graphical output to an existing plot. In particular, the regions and coordinate systems used in the construction of a plot are also available for adding further output to the plot. For example, it is possible to position a text label relative to the scales on the axes of a plot.

3.4.1 Annotating the plot region

Most graphics functions that add output to an existing plot, add the output to the plot region, relative to the user coordinate system.

Graphical primitives

This section describes the graphics functions that provide the most basic graphics output (lines, rectangles, text, etc).

The most common use of this facility is to plot additional sets of data within a plot. The `lines()` function draws lines between `(x, y)` locations, and the `points()` function draws data symbols at `(x, y)` locations. The following code demonstrates a common situation where three different sets of y-values, recorded at the same set of x-values, are plotted together on the same plot (see the top-left plot in Figure 3.16).

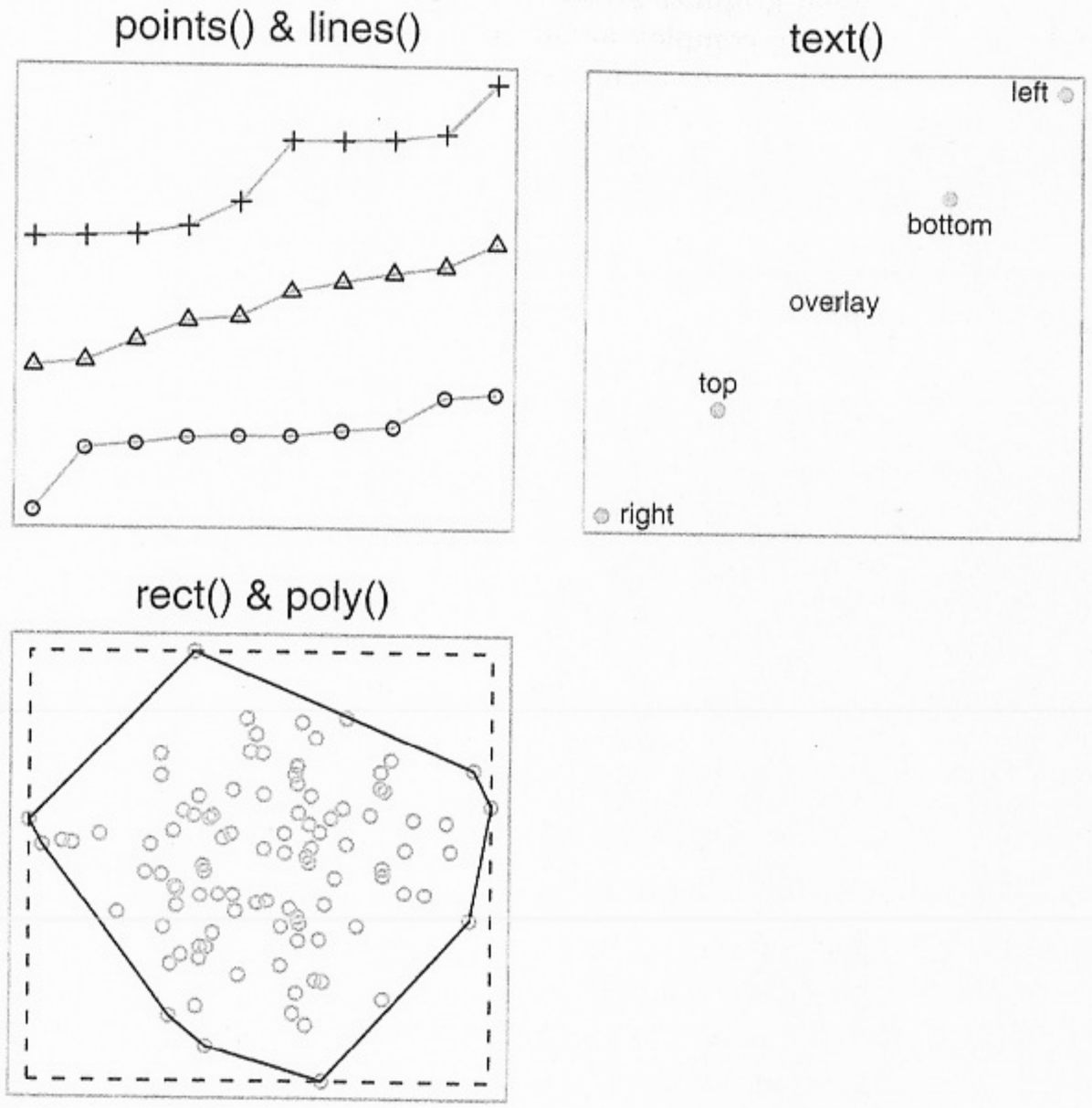


Figure 3.16
Annotating the plot region of a traditional graphics plot. The top-left plot shows points and extra lines being added to an initial line plot. The top-right plot shows text being added to an initial scatterplot. The bottom-left plot shows a dashed rectangle and a polygon being added to an initial scatterplot. Axes and labels have been omitted from the plots in order to avoid clutter.

First some data are generated, consisting of one set of `x` values and three sets of `y` values, and the first set of `y` values are plotted as a grey line (`type="l"` and `col="grey"`).

```
> x <- 1:10
> y <- matrix(sort(rnorm(30)), ncol=3)
> plot(x, y[,1], ylim=range(y), ann=FALSE, axes=FALSE,
       type="l", col="grey")
> box(col="grey")
```

Now a set of points are added for the first set of `y` values, then lines and points are added for the other two sets of `y` values.

```
> points(x, y[,1])
> lines(x, y[,2], col="grey")
> points(x, y[,2], pch=2)
> lines(x, y[,3], col="grey")
> points(x, y[,3], pch=3)
```

The `lines()` function typically draws a single line through many points (though `NA` values in the `(x, y)` locations will create breaks in the line). An alternative is provided by the `segments()` function, which will draw several different straight lines between pairs of end points.

It is also possible to draw text at `(x, y)` locations. This is useful for labeling data locations, particularly using the `pos` argument to offset the text so that it does not overlay any data symbols. The following code creates a diagram demonstrating the use of `text()` (see the top-right plot in Figure 3.16). Again, some data are created and (grey) data symbols are plotted at the `(x, y)` locations.

```
> x <- c(4, 5, 2, 1)
> y <- x
> plot(x, y, ann=FALSE, axes=FALSE, col="grey", pch=16)
> points(3, 3, col="grey", pch=16)
> box(col="grey")
```

Now some text labels are added, with each one offset in a different way from the `(x, y)` location. Notice that the arguments to `text()` may be vectors so that several pieces of text are drawn by the one function call.

```
> text(x, y, c("bottom", "left", "top", "right"), pos=1:4)
> text(3,. 3, "overlay")
```

There are also the functions `rect()` and `polygon()` for drawing rectangles and polygons. The arguments to `rect()` may be vectors, in which case multiple rectangles are drawn. Multiple polygons may be drawn using `polygon()` by inserting an `NA` value between each set of polygon vertexes. R will draw self-intersecting polygons, but does not handle polygons with holes. For both `rect()` and `polygon()`, the `col` argument specifies the color to *fill* the interior of the shape and the argument `border` controls the color of the line around the boundary of the shape. The following code demonstrates the use of these functions. First, data are generated and plotted (as grey circles).

```
> x <- rnorm(100)
> y <- rnorm(100)
> plot(x, y, ann=FALSE, axes=FALSE, col="grey")
> box(col="grey")
```

Now we draw a dashed bounding box for the data using `rect()` and a solid convex hull using `polygon()` (and `chull()` to calculate the hull; see the bottom-left plot of Figure 3.16).

```
> rect(min(x), min(y), max(x), max(y), lty="dashed")
> hull <- chull(x, y)
> polygon(x[hull], y[hull])
```

Like the `plot()` function, the `text()`, `lines()`, and `points()` functions are generic. This means that they have flexible interfaces for specifying `(x, y)` locations, or they produce different output when given objects of a particular class in the `x` argument. For example, both `lines()`, and `points()` will accept formulae for specifying the `(x, y)` locations and the `lines()` function will behave sensibly when given a `ts` (time series) object to draw.

As a parallel to the `matplot()` function (see page 29), there are functions `matpoints()` and `matlines()` specifically for adding lines and data symbols to a plot given `x` or `y` as matrices.

Graphical utilities

In addition to the low-level graphical primitives of the previous section, there are a number of utility functions that provide a set of slightly more complex shapes.

The `grid()` function adds a series of grid lines to a plot. This is simply a series of line segments, but the default appearance (light grey and dotted) is suited to the purpose of providing visual cues to the viewer without interfering with the primary data symbols.

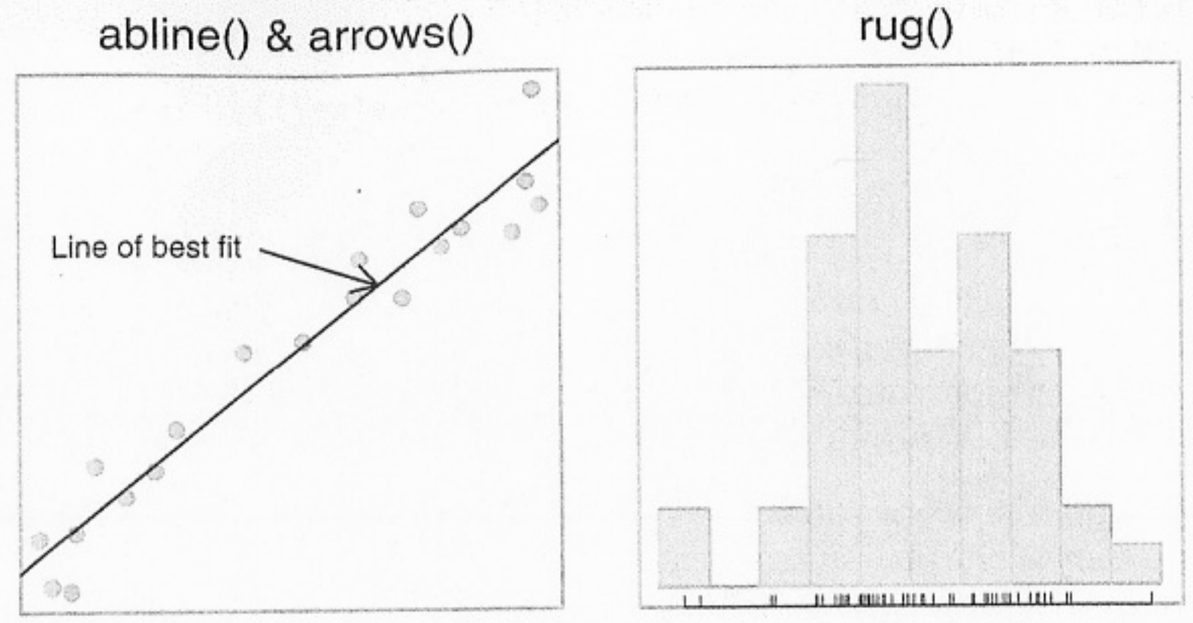


Figure 3.17
More examples of annotating the plot region of a traditional graphics plot. The left-hand plot shows a line of best fit (plus a text label and arrow) being added to an initial scatterplot. The right-hand plot shows a series of ticks being added as a rug plot on an initial histogram.

The `abline()` function provides a number of convenient ways to add a line (or lines) to a plot. The line(s) can be specified either by a slope and y-axis intercept, or as a series of x-locations for vertical lines or y-locations for horizontal lines. The function will also accept the coefficients from a linear regression analysis (even as an `lm` object), thereby providing a simple way to add a line of best fit to a scatterplot.

The `arrows()` function draws line segments and augments them with simple arrowheads at either end. The following code annotates a basic scatterplot with a line and arrows (see the left plot of Figure 3.17).

First, some data are generated and plotted.

```
> x <- runif(20, 1, 10)
> y <- x + rnorm(20)
> plot(x, y, ann=FALSE, axes=FALSE, col="grey", pch=16)
> box(col="grey")
```

Now a line of best fit is drawn through the data using `abline()` and a text label and arrow are added using `text()` and `arrows()`.

```
> lmfit <- lm(y ~ x)
> abline(lmfit)
> arrows(5, 8, 7, predict(lmfit, data.frame(x=7)),
         length=0.1)
> text(5, 8, "Line of best fit", pos=2)
```

The `box()` function draws a rectangle around the boundary of the plot region. The `which` argument makes it possible to draw the rectangle around the current figure region, inner region, or outer region instead. The `box()` function has been used in each of the examples in this section.

The `rug()` function produces a "rug" plot along one of the axes, which consists of a series of tick marks representing data locations. This can be useful to represent an additional one-dimensional plot of data (e.g., in combination with a density curve). The following code uses this function to annotate a histogram (see the right plot of Figure 3.17).

```
> y <- rnorm(50)
> hist(y, main="", xlab="", ylab="", axes=FALSE,
       border="grey", col="light grey")
> box(col="grey")
> rug(y, ticksize=0.02)
```

3.4.2 Missing values and non-finite values

R has special values representing missing observations (`NA`) and non-finite values (`NaN` and `Inf`). Most traditional graphics functions allow such values within `(x, y)` locations and handle them by not drawing the relevant location. For drawing data symbols or text, this means the relevant data symbol or piece of text will not be drawn. For drawing lines, this means that lines to or from the relevant location are not drawn; a gap is created in the line. For drawing rectangles, an entire rectangle will not be drawn if any of the four boundary locations are missing or non-finite.

Polygons are a slightly more complex case. For drawing polygons, a missing or non-finite value in `x` or `y` is interpreted as the end of one polygon and the start of another. Figure 3.18 shows an example. On the left, a polygon is drawn through 12 locations evenly spaced around a circle. On the right, the first, fifth, and ninth locations have been set to `NA` so the output is split into three separate polygons.

Missing or non-finite values can also be specified for some traditional graphics state settings. For example, if a color setting is missing or non-finite then

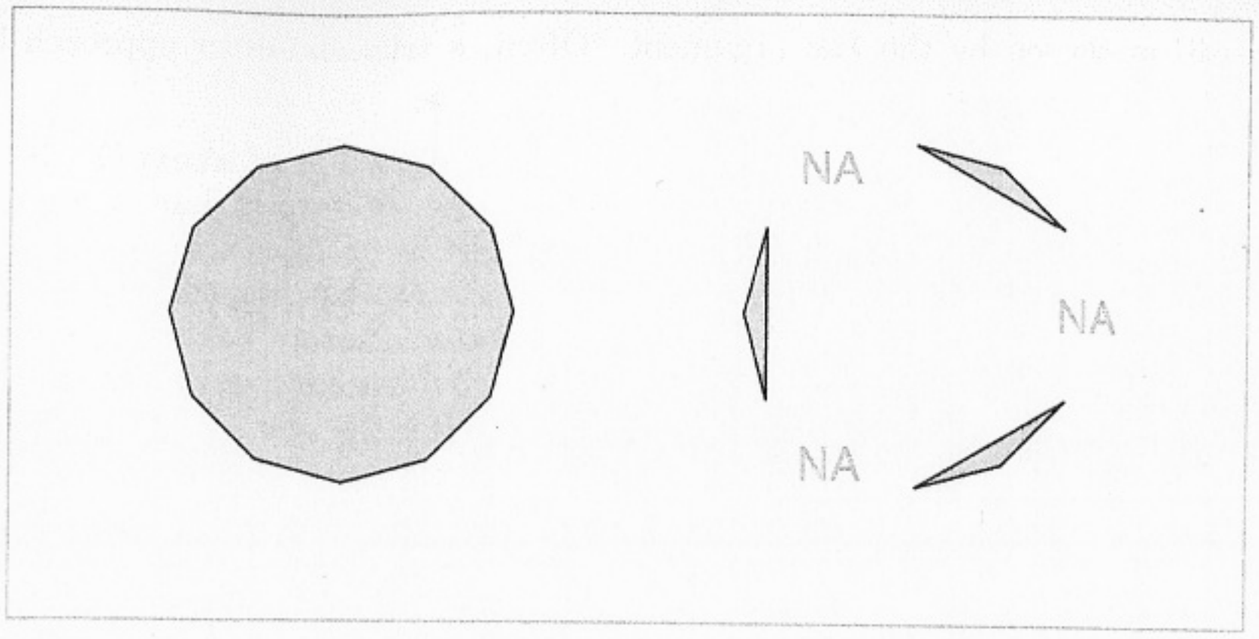


Figure 3.18
Drawing polygons using the `polygon()` function. On the left, a single polygon (dodecagon) is produced from multiple `(x, y)` locations. On the right, the first, fifth, and ninth values have been set to `NA`, which splits the output into three separate polygons.

nothing is drawn (this is a brute-force way to specify a completely transparent color). Similarly, specifying a missing value or non-finite value for `cex` means that the relevant data symbol or piece of text is not drawn.

3.4.3 Annotating the margins

There are only two functions that produce output in the figure or outer margins, relative to the margin coordinate systems (Section 3.1.1).

The `mtext()` function draws text at any location in any of the margins. The `outer` argument controls whether output goes in the figure or outer margins. The `side` argument determines which margin to draw in: `1` means the bottom margin, `2` means the left margin, `3` means the top margin, and `4` means the right margin.

Text is drawn a number of lines of text away from the edges of the plot region for figure margins, or a number of lines away from the edges of the inner region for outer margins. In the figure margins, the location of the text along the margin can be specified relative to the user coordinates on the relevant axis using the `at` argument. In some cases it is possible to specify the location as a proportion of the length of the margin using the `adj` argument, but this is dependent on the value of the `las` state setting. For certain `las` settings, the `adj` argument instead controls the justification of the text relative to a

position chosen by the `las` argument. Often, a trial-and-error approach is required to achieve the desired result.

The `title()` function is essentially a specialized version of `mtext()`. It is more convenient for producing a few specific types of output, but much less flexible than `mtext()`. This function can be used to produce a main title for a plot (in the top figure margin), axis labels (in the left and bottom figure margins), and a sub-title for a plot (in the bottom margin below the x-axis label). The output from this function is heavily influenced by various graphics state settings, such as `cex.main` and `col.main` (for the size and color of the title).

With a little extra effort, it is also possible to produce graphical output in the figure or outer margins using the functions that normally draw in the plot region (e.g., `points()` and `lines()`). In order to do this, the clipping region of the plot must first be set using the `xpd` state setting (see Section 3.2.7). This approach is not very convenient because the functions are drawing relative to user coordinates rather than locations relative to the margin coordinate systems. Nevertheless, it can sometimes be useful.

The following code demonstrates the use of `mtext()` and a simple application of using `lines()` outside the plot region for drawing what appears to be a rectangle extending across two plots (see Figure 3.19).*

First of all, the `mfrow` setting is used to set up an arrangement of two figure regions, one above the other. The clipping region is set to the entire device using `xpd=NA`.

```
> y1 <- rnorm(100)
> y2 <- rnorm(100)
```

```
> par(mfrow=c(2, 1), xpd=NA)
```

The first data set is plotted as a time series on the top plot and a label is added at the left end of figure margin 3. In addition, thick grey lines are drawn to represent the top of the rectangle that deliberately extend well below the bottom of the plot.

*This example was motivated by a question to R-help on December 14 2004 with subject: "drawing a rectangle through multiple plots".

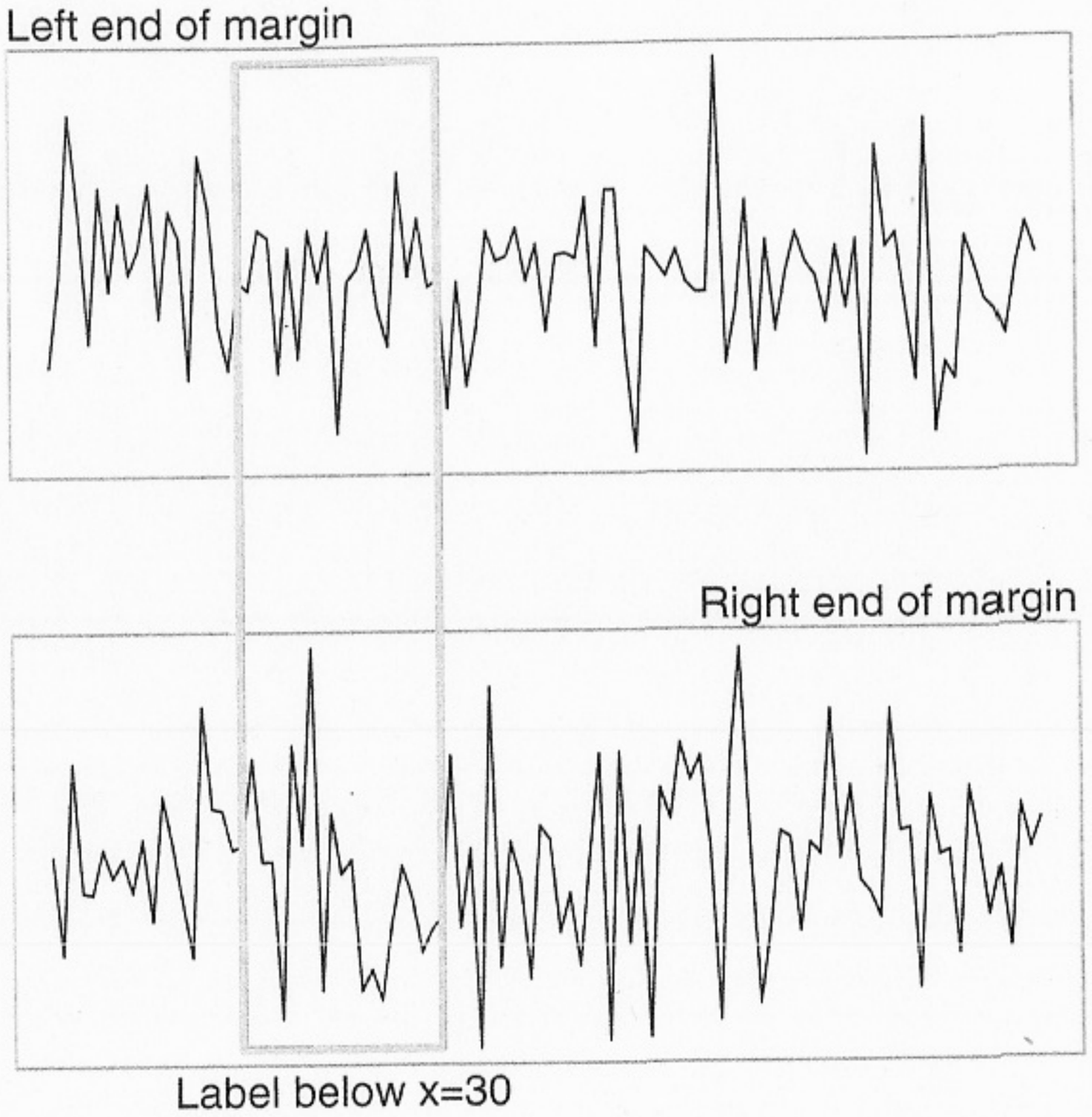


Figure 3.19
Annotating the margins of a traditional graphics plot. Text has been added in margin 3 of the top plot and in margins 1 and 3 in the bottom plot. Thick grey lines have been added to both plots (and overlapped so that it appears to be a single rectangle across the plots).

```
> plot(y1, type="l", axes=FALSE,
       xlab="", ylab="", main="")
> box(col="grey")
> mtext("Left end of margin", adj=0, side=3)
> lines(x=c(20, 20, 40, 40), y=c(-7, max(y1), max(y1), -7),
        lwd=3, col="grey")
```

The second data set is plotted as a time series in the bottom plot, a label is added to this plot at the right end of figure margin 3, and another label is drawn beneath the x-location 30 in figure margin 1. Finally, thick grey lines are drawn to represent the bottom of the rectangle that deliberately extend above the plot. These lines overlap the lines drawn with respect to the top plot to create the impression of a single rectangle traversing both plots.

```
> plot(y2, type="l", axes=FALSE,
       xlab="", ylab="", main="")
> box(col="grey")
> mtext("Right end of margin", adj=1, side=3)
> mtext("Label below x=30", at=30, side=1)
> lines(x=c(20, 20, 40, 40), y=c(7, min(y2), min(y2), 7),
        lwd=3, col="grey")
```

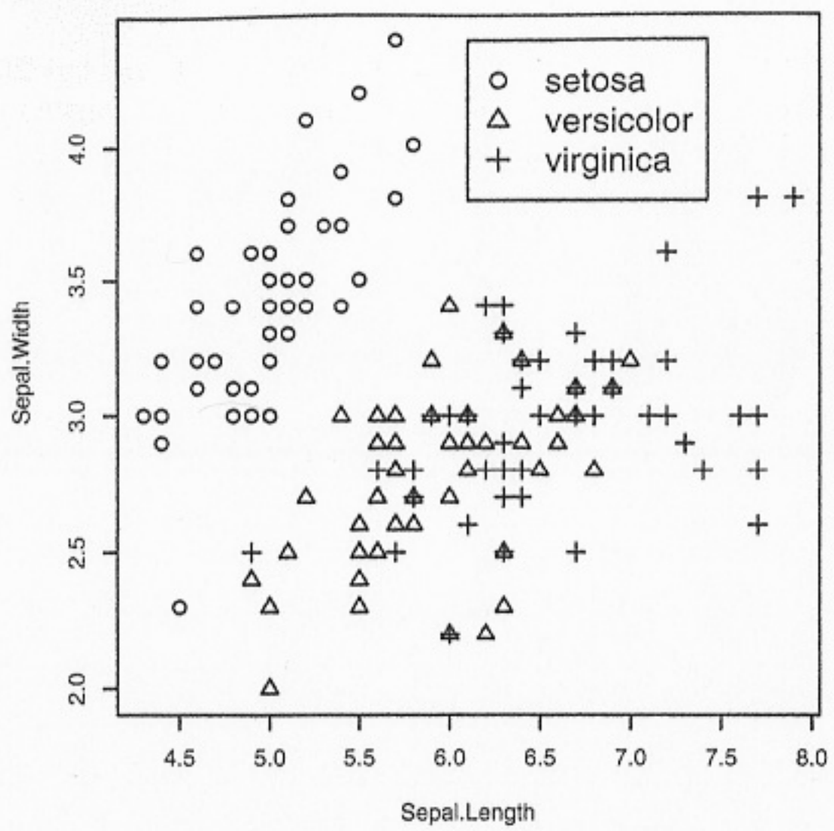
3.4.4 Legends

The traditional graphics system provides the `legend()` function for adding a legend or key to a plot. The legend is usually drawn within the plot region, and is located relative to user coordinates. The function has many arguments, which allow for a great deal of flexibility in the specification of the contents and layout of the legend. The following code demonstrates a couple of typical uses.

The first example shows a scatterplot with a legend to relate group names to different symbols (see the top plot in Figure 3.20).

```
> with(iris,
       plot(Sepal.Length, Sepal.Width,
            pch=as.numeric(Species), cex=1.2))
> legend(6.1, 4.4, c("setosa", "versicolor", "virginica"),
         cex=1.5, pch=1:3)
```

The next example shows a barplot with a legend to relate group names to



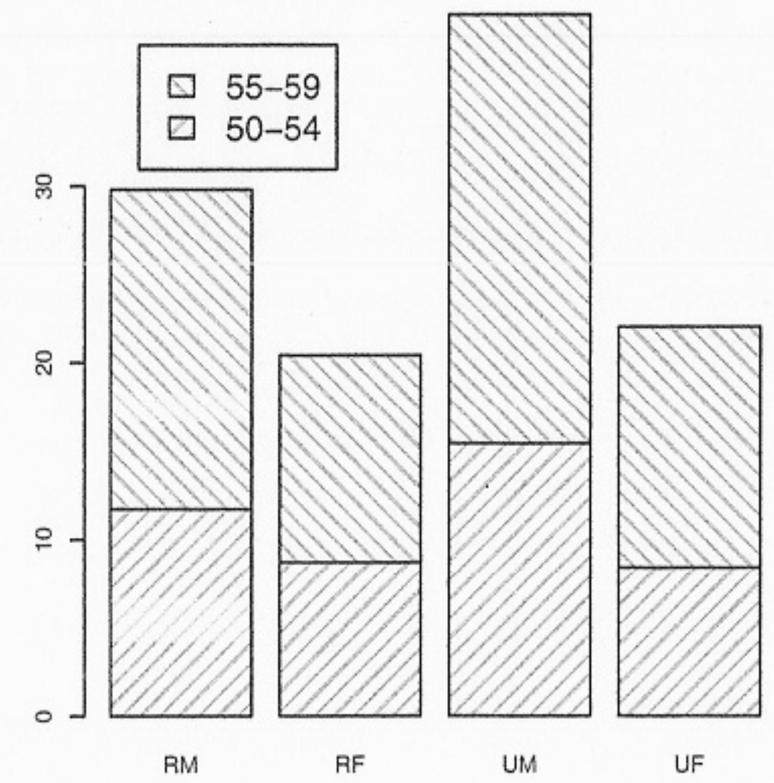


Figure 3.20
Some simple legends. Legends can be added to any kind of plot and can relate text labels to different symbols or different fill colors or patterns.

different fill patterns (see the bottom plot in Figure 3.20).*

```
> barplot(VADeaths[1:2,], angle=c(45, 135), density=20,
          col="grey", names=c("RM", "RF", "UM", "UF"))
> legend(0.4, 38, c("55-59", "50-54"), cex=1.5,
         angle = c(135, 45), density = 20, fill = "grey")
```

It should be noted that it is entirely the responsibility of the user to ensure that the legend corresponds to the plot. There is no automatic checking that data symbols in the legend match those in the plot, or that the labels in the legend have any correspondence with the data.

Some high-level functions draw their own legend specific to their purpose (e.g., `filled.contour()`).

3.4.5 Axes

In most cases, the axes that are automatically generated by the traditional graphics system will be sufficient for a plot. This is true even when the data being plotted on an axis are non-numeric. For example, the axes of a boxplot or barplot are labeled appropriately using group names.

Section 3.2.5 describes ways in which the default appearance of automatically-generated axes can be modified, but it is more often the case that the user needs to inhibit the production of the automatic axis and draw a customized axis using the `axis()` function.

The first step is to inhibit the default axes. Most high-level functions should provide an `axes` argument which, when set to `FALSE`, indicates that the high-level function should not draw axes. Specifying the traditional graphics setting `xaxt="n"` (or `yaxt="n"`) may also do the trick.

The `axis()` function can draw axes on any side of a plot (chosen by the `side` argument), and the user can specify the location along the axis of tick marks and the text to use for tick labels (using the `at` and `labels` arguments respectively). The following code demonstrates a simple example of a plot where the automatic axes are inhibited and custom axes are drawn, including a "secondary" y-axis on the right side of the plot (see Figure 3.21).

First of all, some temperature data are generated and an empty plot is created with no data symbols and no axes.

*The data for the scatterplot are from the `iris` data set (see page 29) and the data for the histogram are from the `VADeaths` data set (see page 3).

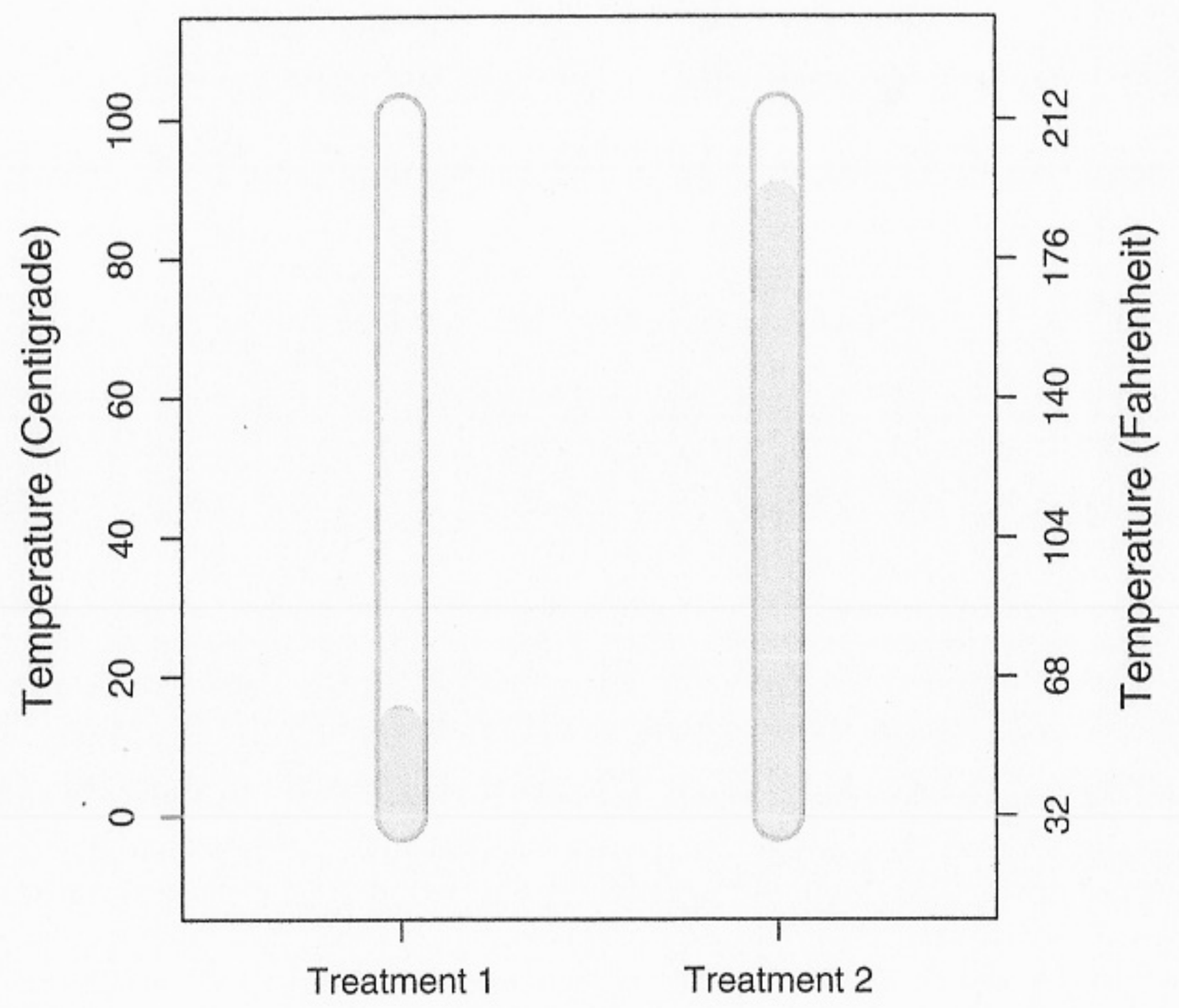


Figure 3.21
Customizing axes. An initial plot is drawn with a y-scale in degrees centigrade, then a secondary y-axis is drawn with a scale in degrees Fahrenheit. The x-axis is drawn using special text labels, rather than the default numeric locations of the tick marks.

```
> x <- 1:2
> y <- runif(2, 0, 100)
> par(mar=c(4, 4, 2, 4))
> plot(x, y, type="n", xlim=c(0.5, 2.5), ylim=c(-10, 110),
       axes=FALSE, ann=FALSE)
```

Next, the main y-axis is drawn with specific tick locations to represent the Centigrade scale.

```
> axis(2, at=seq(0, 100, 20))
> mtext("Temperature (Centigrade)", side=2, line=3)
```

Now the bottom axis is drawn with special labels and a secondary y-axis is drawn to represent the Fahrenheit scale.

```
> axis(1, at=1:2, labels=c("Treatment 1", "Treatment 2"))
> axis(4, at=seq(0, 100, 20), labels=seq(0, 100, 20)*9/5 + 32)
> mtext("Temperature (Fahrenheit)", side=4, line=3)
> box()
```

Finally, some thermometer-like symbols are drawn to represent the actual temperatures.

```
> segments(x, 0, x, 100, lwd=20, col="dark grey")
> segments(x, 0, x, 100, lwd=16, col="white")
> segments(x, 0, x, y, lwd=16, col="light grey")
```

The `axis()` function is not generic, but there are special alternative functions for plotting time related data. The functions `axis.Date()` and `axis.POSIXct()` take an object containing dates and produce an axis with appropriate labels representing times, days, months, and years (e.g., `10:15`, `Jan 12` or `1995`).

In some cases, it may be useful to draw tick marks at the locations that the default axis would use, but with different labels. The `axTicks()` function can be used to calculate these default locations. This function is also useful for enforcing an `xaxp` (or `yaxp`) graphics state setting. If these settings are specified via `par()`, they usually have no effect because the traditional graphics system almost always calculates the settings itself. The user can choose these settings by passing them as arguments to `axTicks()`, then passing the resulting locations via the `at` argument to `axis()`.

3.4.6 Mathematical formulae

Any R graphics function that draws text should accept both a normal string, e.g., `"some text"`, and an R expression, which is typically the result of a call to the `expression()` function. If an expression is specified as the text to draw, then it is interpreted as a mathematical formula and is formatted appropriately. This section provides some simple examples of what can be achieved. For a complete description of the available features, type `help(plotmath)` or `demo(plotmath)` in an R session.*

When an R expression is provided as text to draw in graphical output, the expression is evaluated to produce a mathematical formula. This evaluation is very different from the normal evaluation of R expressions: certain names are interpreted as special mathematical symbols, e.g., `alpha` is interpreted as the Greek symbol α; certain mathematical operators are interpreted as literal symbols, e.g., `a + ` is interpreted as a plus sign symbol; and certain functions are interpreted as mathematical operators, e.g., `sum(x, i==1, n)` is interpreted as $\sum_{i=1}^{n} x$. Figure 3.22 shows some examples of expressions and the output that they create.

In some situations, for example, when calling graphics functions from within a loop, or when calling graphics functions from within another function, the expression representing the mathematical formula must be constructed using values within variables as well as literal symbols and constants. A variable name within an expression will be treated as a literal symbol (i.e., the variable name will be drawn, not the value within the variable). The solution in such cases is to use the `substitute()` function to produce an expression. The following code shows the use of `substitute()` to produce a label where the year is stored in a variable.

```
> myfunction <- function(year) {
    text(0.5, 0.5, substitute(paste("Temperature (",
                                    degree, "C) in ", year),
         list(year=year)))
  }
```

The mathematical annotation feature makes use of information about the dimensions of individual characters to perform the formatting of the formula. For some output formats, such information is not available, so mathematical formulae cannot be produced. However, mathematical formulae are supported on the major screen devices (X11, Windows, and Quartz) and information

*Further information can also be obtained from an article in the Journal of Computational and Graphical Statistics[45].

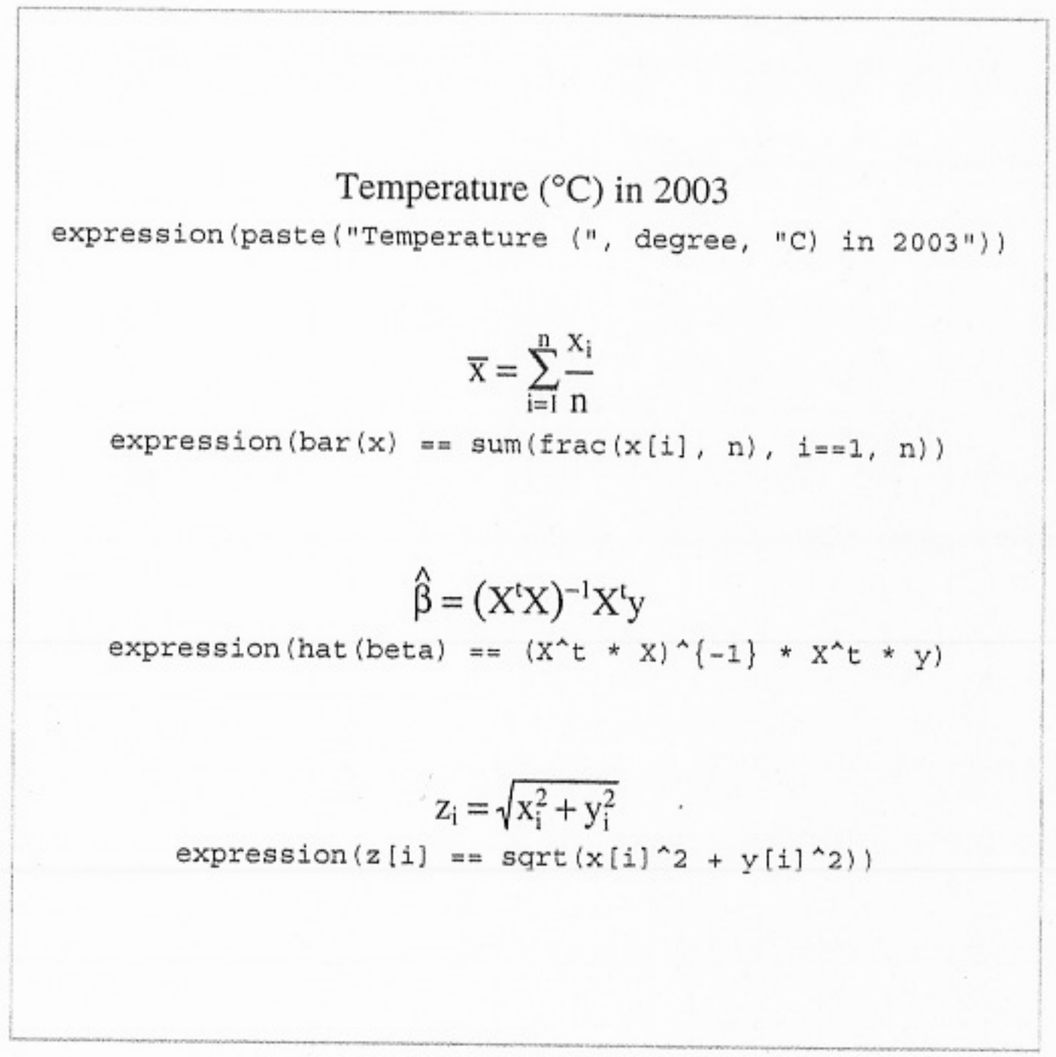


Figure 3.22
Mathematical formulae in plots. For each example, the output is shown in a serif font, and below that, in a typewriter font, is the R expression required to produce the output.

for the standard Adobe Type 1 fonts is distributed with R so mathematical formulae should always be available for PostScript and PDF output.

3.4.7 Coordinate systems

The traditional graphics system provides a number of coordinate systems for conveniently locating graphical output (see Section 3.1.1). Graphical output in the plot region is automatically positioned relative to the scales on the axes and text in the figure margins is placed in terms of a number of lines away from the edge of the plot (i.e., a scale that naturally corresponds to the size of the text).

It is also possible to locate output according to other coordinate systems that are not automatically supplied, but a little more work is required by the user. The basic principle is that the traditional graphics state can be queried to determine features of existing coordinate systems, then new coordinate systems can be calculated from this information.

The par function

As well as being used to enforce new graphics state settings, the function `par()` can also be used to query current graphics state settings. The most useful settings are: `din`, `fin`, and `pin`, which reflect the current size, `(width, height)`, of the graphics device, figure region, and plot region, in inches; and `usr`, which reflects the current user coordinate system (i.e., the ranges on the axes). The values of `usr` are in the order `(xmin, xmax, ymin, ymax)`. When a scale has a logarithmic transformation, the values are `(10^xmin, 10^xmax, 10^ymin, 10^ymax)`.

There are also settings that reflect the size, `(width, height)`, of a "standard" character. The setting `cin` gives the size in inches, `cra` in "rasters" or pixels, and `cxy` in "user coordinates." However, these values are not very useful because they only refer to a `cex` value of `1` (i.e., they ignore the current `cex` setting) *and* they only refer to the `ps` value when the current graphics device was first opened. Of more use are the `strheight()` function and the `strwidth()` function. These calculate the height and width of a given piece of text in inches, or in terms of user coordinates, or as a proportion of the current figure region (taking into account the current `cex` and `ps` settings).

The following code demonstrates a simple example of making use of customized coordinates where a ruler is drawn showing centimeter units (see Figure 3.23).

A blank plot region is set up first and calculations are performed to establish

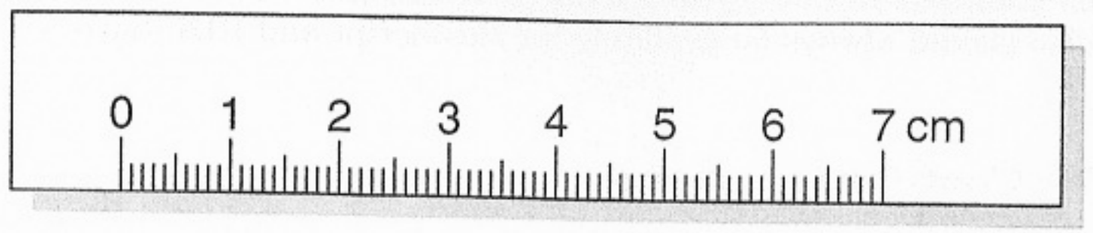


Figure 3.23
Custom coordinate systems. The lines and text are drawn relative to real physical centimeters (rather than the default coordinate system defined by the scales on plot axes).

the relationship between user coordinates in the plot and physical centimeters.*

```
> plot(0:1, 0:1, type="n", axes=FALSE, ann=FALSE)
> usr <- par("usr")
> pin <- par("pin")
> xcm <- diff(usr[1:2])/(pin[1]*2.54)
> ycm <- diff(usr[3:4])/(pin[2]*2.54)
```

Now drawing can occur with positions expressed in terms of centimeters. First of all a "drop shadow" is drawn to give a three-dimensional effect by drawing a grey rectangle offset by 2mm from the main ruler. The call to `par()` makes sure that the grey rectangle is not clipped to the plotting region (see Section 3.2.7).

```
> par(xpd=NA)
> rect(0 + 0.2*xcm, 0 - 0.2*ycm,
       1 + 0.2*xcm, 1 - 0.2*ycm,
       col="grey", border=NA)
```

The ruler itself is drawn with a call to `rect()` to draw the edges of the ruler, a call to `segments()` to draw the scale, and calls to `text()` to label the scale.

*R graphics relies on a graphics device providing accurate information on the physical size of the natural units on the device (e.g., the physical size of pixels on a computer screen). If a graphics device does not give accurate information, when R attempts to draw output with an physical size (e.g., a line 1 inch long), it may not appear with the exact physical size on the device. The physical size of output for PostScript and PDF files should always be correct, but small inaccuracies may occur when specifying output with an physical size (such as inches) on screen devices such as Windows and X11 windows.


```
> rect(0, 0, 1, 1, col="white")
> segments(seq(1, 8, 0.1)*xcm, 0,
           seq(1, 8, 0.1)*xcm,
           c(rep(c(0.5, rep(0.25, 4),
                   0.35, rep(0.25, 4)),
                 7), 0.5)*ycm)
> text(1:8*xcm, 0.6*ycm, 0:7, adj=c(0.5, 0))
> text(8.2*xcm, 0.6*ycm, "cm", adj=c(0, 0))
```

There are utility functions, `xinch()` and `yinch()`, for performing the inches-to-user coordinates transformation (plus `xyinch()` for converting a location in one step and `cm()` for converting inches to centimeters).

One problem with performing coordinate transformations like these is that the locations and sizes being drawn have no memory of how they were calculated. They are specified as locations and dimensions in user coordinates. This means that if the device is resized (so that the relationship between physical dimensions and user coordinates changes), the locations and sizes will no longer have their intended meaning. If, in the above example, the device is resized, the ruler will no longer accurately represent centimeter units. This problem will also occur if output is copied from one device to another device that has different physical dimensions. The `legend()` function performs calculations like these when arranging the components of a legend and its output is affected by device resizes and copying between devices.*

Overlaying output

It is sometimes useful to plot two data sets on the same plot where the data sets share a common x-variable, but have very different y-scales. This can be achieved in at least two ways. One approach is simply to use `par(new=TRUE)` to overlay two distinct plots on top of each other (care must be taken to avoid conflicting axes overwriting each other). Another approach is to explicitly reset the `usr` state setting before plotting a second set of data. The following code demonstrates both approaches to produce exactly the same result (see the top plot of Figure 3.24).

The data are yearly numbers of drunkenness-related arrests† and mean annual temperature in New Haven, Connecticut from 1912 to 1971. The temperature

*It is possible to work around these problems in R version 2.1.0 and above by using the `recordGraphics()` function, although this function should be used with extreme care.

†These data were obtained from "Crime Statistics and Department Demographics" on the New Haven Police Department Web Site:
`http://www.cityofnewhaven.com/police/html/stats/crime/yearly/1863-1920.htm`

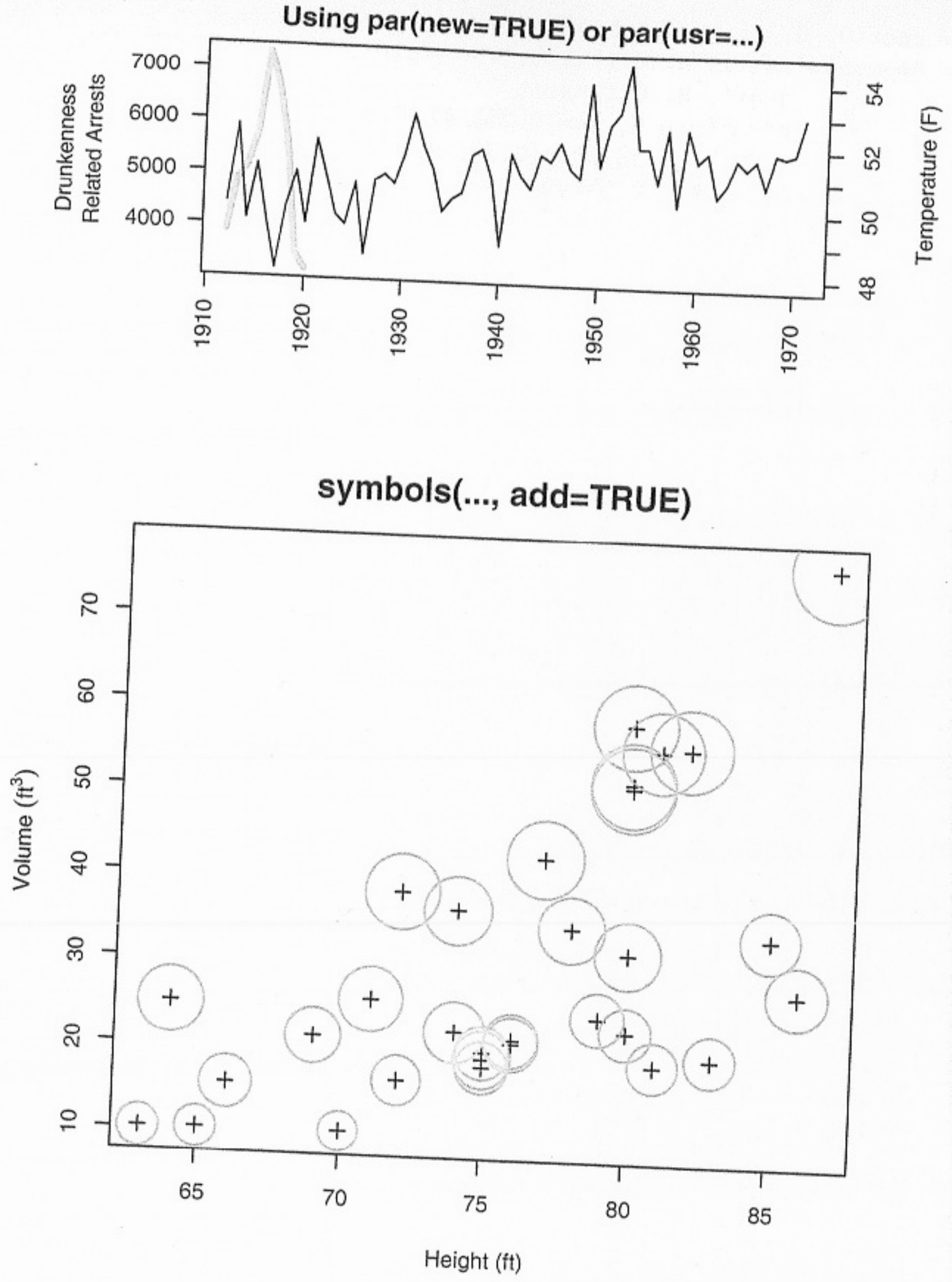


Figure 3.24
Overlaying plots. In the top plot, two line plots are drawn one on top of each other to produce aligned plots of two data sets with very different scales. In the bottom plot, the plotting function `symbols()` is used in "annotating mode" so that it adds circles to an existing scatterplot rather than producing a complete plot itself.

data are available as the data set `nhtemp` in the `datasets` package. There are only arrests data for the first 9 years.

```
> drunkenness <- ts(c(3875, 4846, 5128, 5773, 7327,
                      6688, 5582, 3473, 3186,
                      rep(NA, 51)),
                    start=1912, end=1971)
```

The first approach is to draw a plot of the drunkenness data, call `par(new=TRUE)`, then draw a complete second plot of the temperature data on top of the first plot. The second plot does not draw default axes (`axes=FALSE`), but uses the `axis()` function to draw a secondary y-axis to represent the temperature scale.

```
> par(mar=c(5, 6, 2, 4))
> plot(drunkenness, lwd=3, col="grey", ann=FALSE, las=2)
> mtext("Drunkenness\nRelated Arrests", side=2, line=3.5)
> par(new=TRUE)
> plot(nhtemp, ann=FALSE, axes=FALSE)
> mtext("Temperature (F)", side=4, line=3)
> title("Using par(new=TRUE)")
> axis(4)
```

The second approach draws only one plot (for the drunkenness data). The user coordinate system is then redefined by specifying a new `usr` setting and the second "plot" is produced simply using `lines()`. Again, a secondary axis is drawn using the `axis()` function.

```
> par(mar=c(5, 6, 2, 4))
> plot(drunkenness, lwd=3, col="grey", ann=FALSE, las=2)
> mtext("Drunkenness\nRelated Arrests", side=2, line=3.5)
> usr <- par("usr")
> par(usr=c(usr[1:2], 47.6, 54.9))
> lines(nhtemp)
> mtext("Temperature (F)", side=4, line=3)
> title("Using par(usr=...)")
> axis(4)
```

Some high-level functions (e.g., `symbols()` and `contour()`) provide an argument called `add` which, if set to `TRUE`, will add the function output to the current plot, rather than starting a new plot. The following code shows the `symbols()` function being used to annotate a basic scatterplot (see the bottom plot of Figure 3.24). The data used in this example are from the `trees` data set (see page 35).

Finally, a horizontal line is drawn to indicate the y-value cut-off, and axes are added to the plot (see the bottom-right plot of Figure 3.25).

```
> abline (h=hline,col="grey")
> box()
> axis(1)
> axis(2)
```

3.4.8 Bitmap images

The R graphics engine has no internal support for drawing bitmaps. Despite this, bitmap images can be represented by drawing a rectangle for each pixel in the image. A convenient interface for this approach is provided by functions in the `pixmap` package[8].

The plot in Figure 3.26 shows an example of what can be achieved using the functions in the `pixmap` package. This plot shows the relationship between the time of day that every second low tide occurred and the phase of the moon, for the port of Auckland, New Zealand in February 2005. The `addlogo()` function has been used to add a bitmap of the moon as a dramatic backdrop for the main plot (the code is not shown, but it is available on the web site for this book). This approach is most appropriate for producing images on screen or in some sort of bitmap format such as PNG. When used for creating vector formats such as PostScript and PDF, the file size grows very rapidly with the size of the bitmap (e.g., the PostScript file for the printed version of Figure 3.26 is more than 5MB!).

3.4.9 Special cases

Some high-level functions are a little more difficult to annotate than others because the plotting regions that they set up either are not immediately obvious, or are not available after the function has run. This section describes a number of high-level functions where additional knowledge is required to perform annotations.

Obscure scales on axes

It is not immediately obvious how to add extra annotation to a barplot or a boxplot in traditional R graphics because the scale on the categorical axis is not obvious.

The difficulty with the `barplot()` function is that because the scale on the

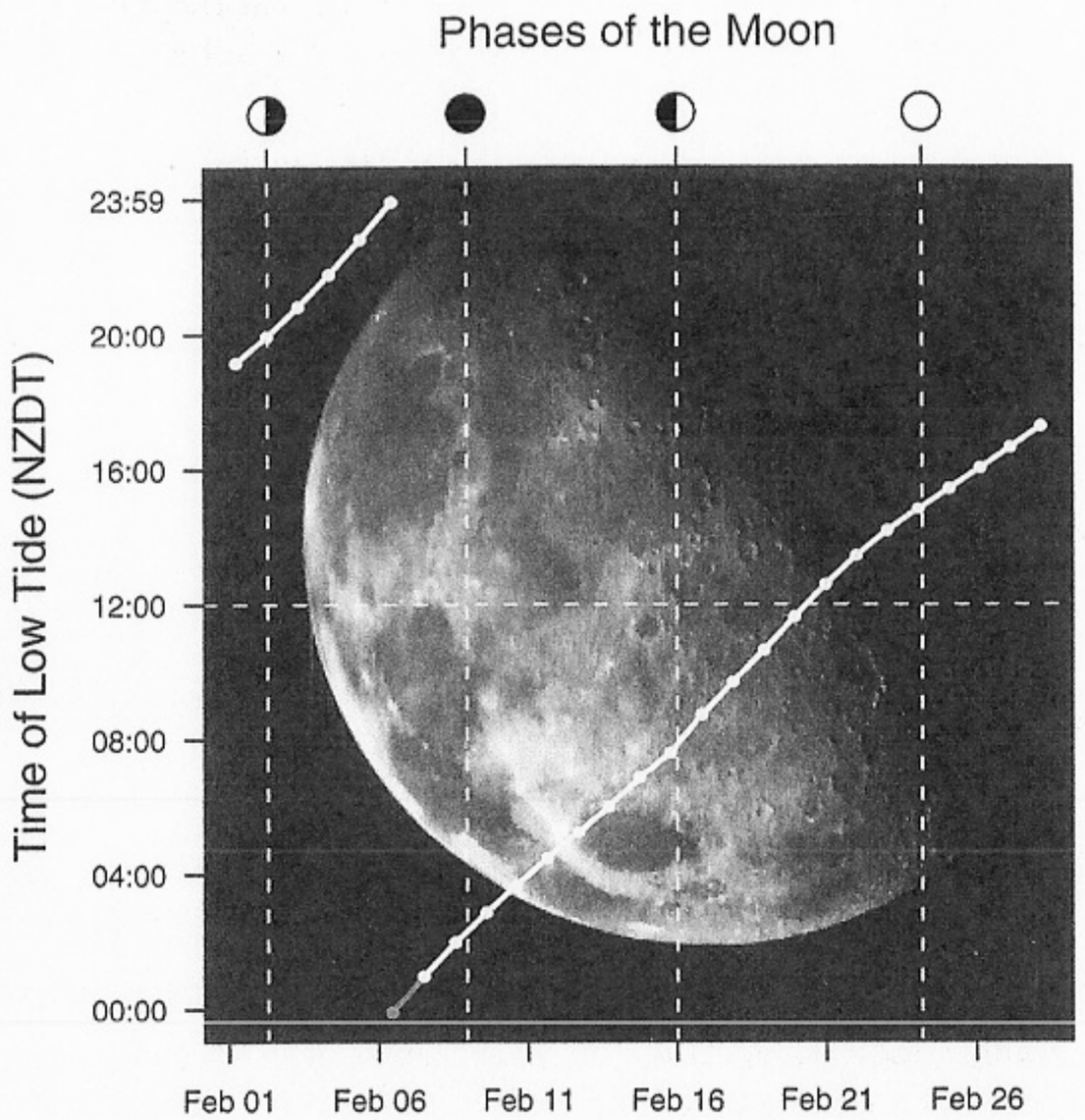


Figure 3.26
Adding a bitmap to a plot. A plot with a bitmap of the moon as a backdrop, added using the `pixmap` package. The bitmap is a view of the Moon's north pole assembled from images taken by the Galileo spacecraft, courtesy of NASA (`image #: PIA00130`). The data on low tides and phases of the moon for Auckland in February 2005 were obtained from Land Information New Zealand (`http://hydro.linz.govt.nz`).

x-axis is not labelled at all by default. the numeric scale is not obvious (and calling `par("usr")` is not much help because the scale that the function sets up is not intuitive either). In order to add annotations sensibly to a barplot it is necessary to capture the value returned by the function. This return value gives the x-locations of the mid-points of each bar that the function has drawn. These midpoints can then be used to locate annotations relative to the bars in the plot.

The code below shows an example of adding extra horizontal reference lines to the bars of a barplot. The mid-points of the bars are saved to a variable called `midpts`, then locations are calculated from those mid-points (and the original counts) to draw horizontal white line segments within each bar using the `segments()` function (see the left plot of Figure 3.27).

```
> y <- sample(1:10)
> midpts <- barplot(y, col=" light grey")
> width <- diff(midpts[1:2])/4
> left <- rep(midpts, y - 1) - width
> right <- rep(midpts, y - 1) + width
> heights <- unlist(apply(matrix(y, ncol=10),
                          2, seq))[-cumsum(y)]
> segments(left, heights, right, heights,
           col="white")
```

The `boxplot()` function is similar to the `barplot()` function in that the x-scale is typically labelled with category names so the numeric scale is not obvious from looking at the plot. Fortunately, the scale set up by the `boxplot()` function is much more intuitive. The individual boxplots are drawn at x-locations `1:n`, where `n` is the number of boxplots being drawn.

The following code provides a simple example of annotating boxplots to add a jittered dotplot of individual data points on top of the boxplots. This provides a detailed view of the data as well as showing the main features via the boxplot. It is also a useful way to show how interesting features of the data, such as small clusters of points, can be hidden by a boxplot. In this example, the jittered data are centered upon the x-locations `1:2` to correspond to the centers of the relevant boxplots (see the right plot of Figure 3.27).*

*The data used in this example are from the `ToothGrowth` data set (see page 3).

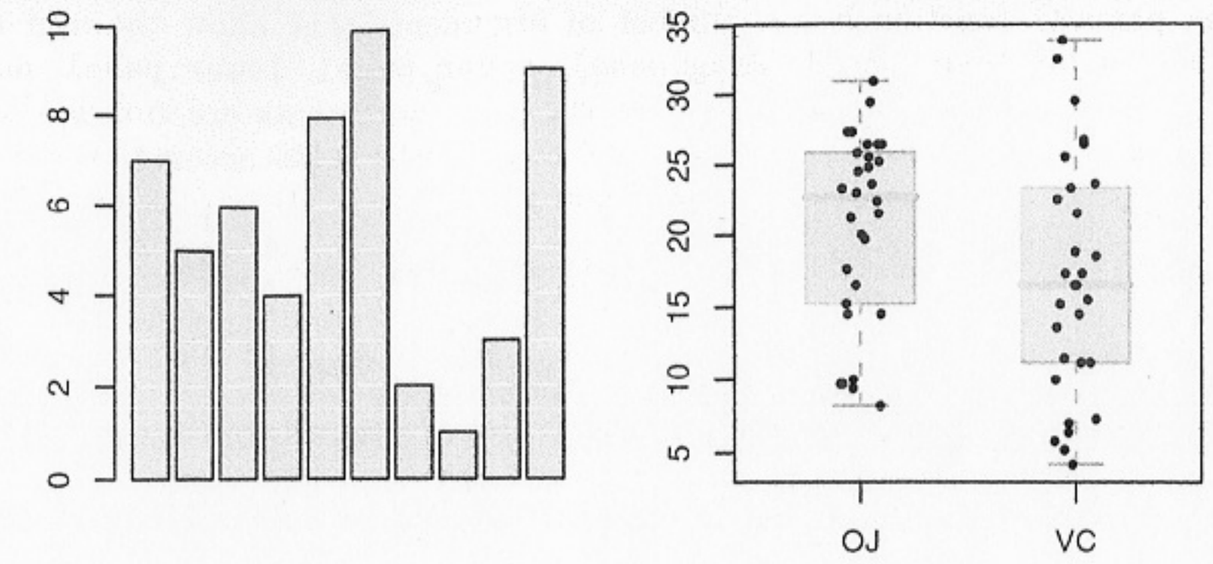


Figure 3.27
Special-case annotations. Some examples of functions where annotation requires special care. In the barplot at left, the value returned by the `barplot()` function is used to add horizontal white lines within the bars. Jittered points are added to the boxplot (right) using the knowledge that the ith box is located at position i on the x-axis.

```
> with(ToothGrowth,
       {
         boxplot(len ~ supp, border="grey",
                 col="light grey", boxwex=0.5)
         points(jitter(rep(1:2, each=30), 0.5),
                unlist(split(len, supp)),
                cex=0.5, pch=16)
       })
```

Functions that draw several plots

The `pairs()` function is an example of a high-level function that draws more than one plot. This function draws a matrix of scatterplots. Such functions tend to save the traditional graphics state before drawing, call `par(mfrow)` or `layout()` to arrange the individual plots, and restore the traditional graphics state once all of the individual plots have been drawn. This means that it is not possible to annotate any of the plots drawn by the `pairs()` function once the function has completed drawing. The regions and coordinate systems that the function set up to draw the individual plots have been thrown away. The only way to annotate the output from such functions is by way of "panel" functions.

The `pairs()` function has a number of arguments that allow the user to specify a function: `panel`, `diag.panel`, `upper.panel`, `lower.panel`, and `text.panel`. The functions specified via these arguments are run as each individual plot is drawn. In this way, the panel function has access to the plot regions that are set up for each individual plot.

The `filled.contour()` function and the `coplot()` function have the same problem as `pairs()`, as the legends that they draw are actually separate plots. Again, they allow annotation via panel function arguments.

The following code demonstrates a simple use of a panel function with the `coplot()` function. The main conditioning plot shows the locations of earthquakes in the Pacific Ocean near Fiji since 1964,* available as the `quakes` data set in the `datasets` package. There are multiple panels, each of which shows the earthquakes that occurred at a particular range of depths. A panel function is specified via the `panel` argument to add maps of Fiji and New Zealand to each panel of `coplot()` output (see Figure 3.28).

The panel function first calls the `rect()` function to overlay a white background and hide the default grid lines. Next, the panel function calls the `points()` function to draw the points that would normally be drawn, but uses a custom plotting symbol (a very small dot). The `map()` function is called to draw the maps of Fiji and the top of the North Island of New Zealand, and the `text()` function is used to add country names. The map is drawn using the `map()` function from the `maps` package.

```
> library(maps)
> coplot(lat ~ long | depth, data = quakes, number=4,
         panel=function(x, y, ...) {
           usr <- par("usr")
           rect(usr[1], usr[3], usr[2], usr[4], col="white")
           map("world2", regions=c("New Zealand", "Fiji"),
               add=TRUE, lwd=0.1, fill=TRUE, col="grey")
           text(180, -13, "Fiji", adj=1, cex=0.7)
           text(170, -35, "NZ", cex=0.7)
           points(x, y, pch=".")
         })
```

There is a predefined panel function called `panel.smooth()`, which draws points and then adds a smoothed line through the points.

*The data were obtained from the Harvard PRIM-H project, who obtained it from Dr. John Woodhouse, Dept. of Geophysics, Harvard University.

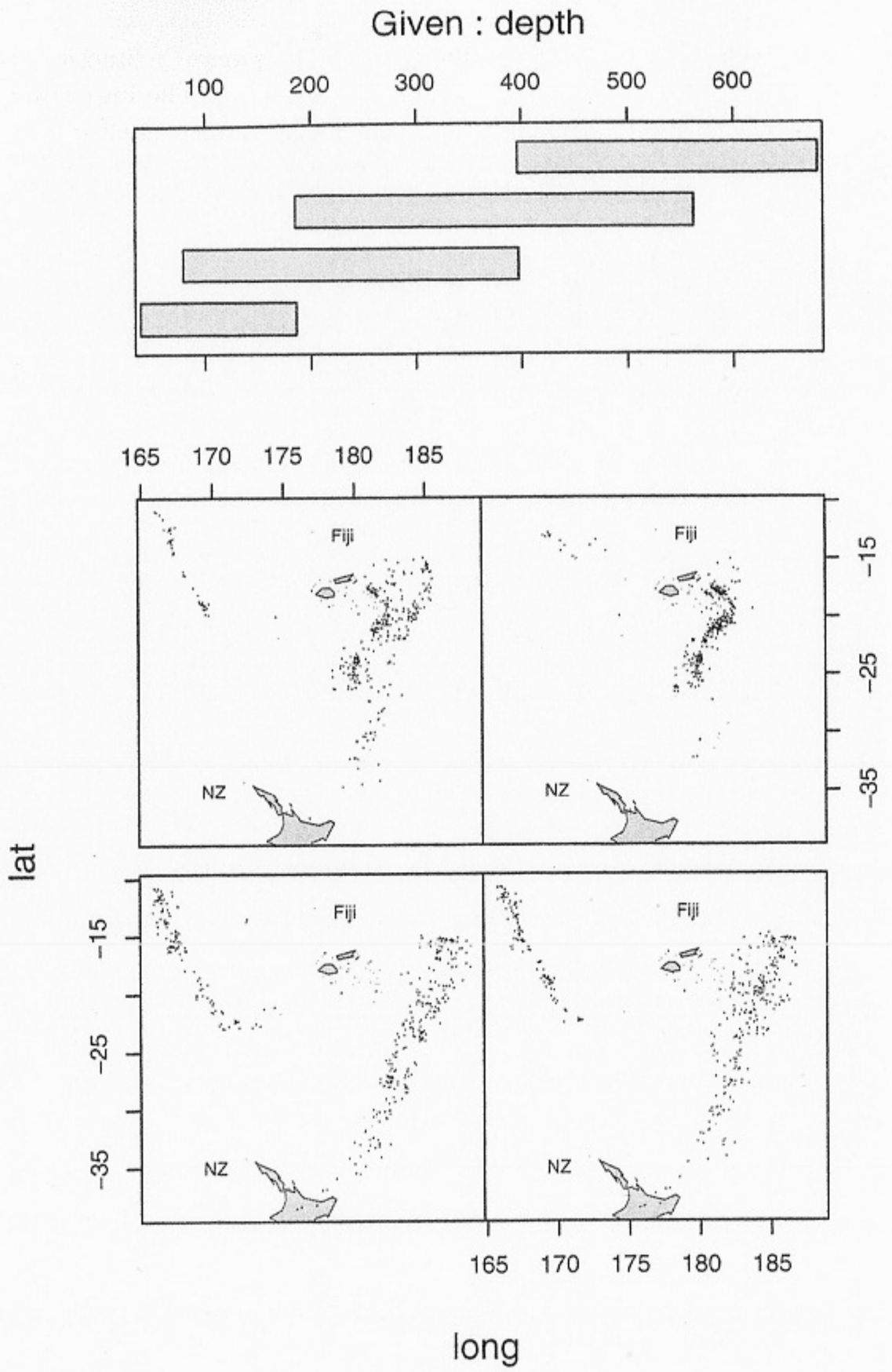


Figure 3.28
A panel function example. An example of using a panel function to add customized output to each element of a multi-panel plot. A panel function is defined that adds maps of Fiji and New Zealand to each panel.

3D plots

It is possible to annotate a plot produced using the `persp()` function, but it is more difficult than for most other high-level functions. The important step is to acquire the transformation matrix that the `persp()` function returns. This can be used to transform 3D locations into 2D locations that can be given to the standard annotation functions such as `lines()` and `text()`. The `persp()` function also has an `add` argument, which allows multiple `persp()` plots to be over-plotted.

The following code demonstrates annotation of `persp()` output to add an indication of the summit and access roads to a plot of the Maunga Whau volcano in Auckland New Zealand (see Figure 3.29).*

The first step is to draw the volcano itself and record the 3D transformation matrix in the variable `trans`.

```
> z <- 2 * volcano
> x <- 10 * (1:nrow(z))
> y <- 10 * (1:ncol(z))
> trans <- persp(x, y, z, theta = 135, phi = 30,
                 scale = FALSE, ltheta = -120,
                 box = FALSE)
> box(col="grey", lwd=1)
```

Now a function is defined that uses the transformation matrix to convert 3D locations into 2D locations relative to the existing plot.

```
> trans3d <- function(x,y,z,pmat) {
    tmat <- cbind(x,y,z,1)%*% pmat
    tmat[,1:2] / tmat[,4]
  }
```

The next code makes use of the transformation function to draw a dot at the summit of the volcano and a text label above that.

```
> summit <- trans3d(x[20], y[31], max(z), trans)
> points(summit[1], summit[2], pch=16)
> summitlabel <- trans3d(x[20], y[31], max(z) + 50, trans)
> text(summitlabel[1], summitlabel[2], "Summit")
```

*The data are from the `volcano` data set (see page 35) and from the `volcano.accessRoad` `volcano.upDownRoad` `volcano.summitRoad` data sets from the `RGraphics` package.

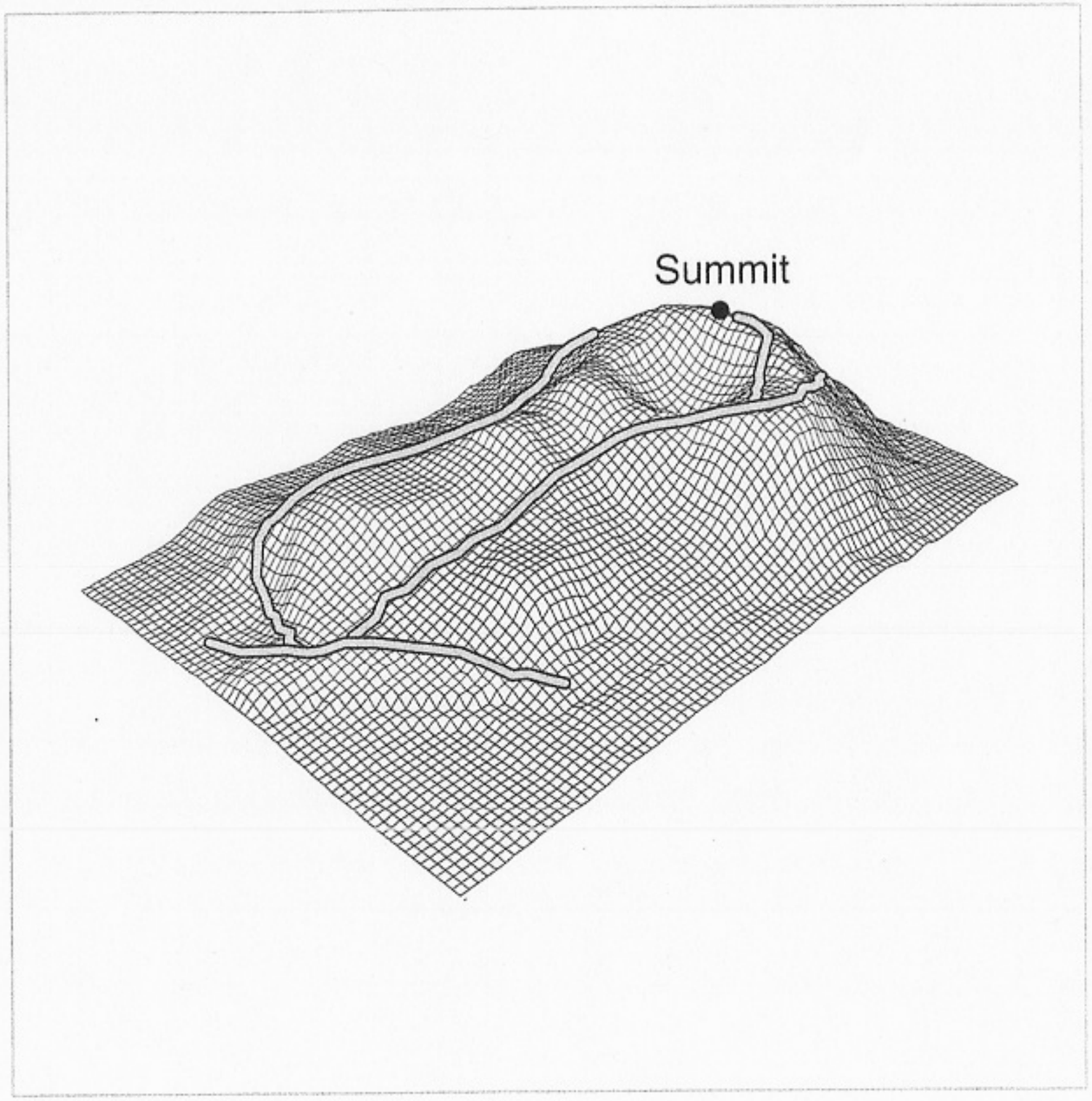


Figure 3.29
Annotating a 3D surface created by `persp()`. Extra points, text, and lines are added to the 3D plot using the transformation matrix returned by the `persp()` function.

Finally, the transformation function is also used to draw lines representing the roads that provide access to the summit of the volcano.

```
> drawRoad <- function(x, y, z, trans) {
    road <- trans3d(x, y, z, trans)
    lines(road[,1], road[,2], lwd=5)
    lines(road[,1], road[,2], lwd=3, col="grey")
  }
> with(volcano.summitRoad,
       drawRoad(srx, sry, srz, trans))
> with(volcano.upDownRoad,
       {
         clipudx <- udx
         clipudx[udx < 230 & udy < 300 |
                 udx < 150 & udy > 300] <- NA
         drawRoad(clipudx, udy, udz, trans)
       })
> with(volcano.accessRoad,
       drawRoad(arx, ary, arz, trans))
```

This example does demonstrate one of the limitations for annotating `persp()` output, namely that there is no support for automatically hiding output that should not be seen. For example, the drawing of the `upDownRoad` has been manually clipped (see the lines involving the variable `clipudx`) in order to avoid drawing the part of the road that should be obscured because it is behind the main cone of the volcano.

3.5 Creating new plots

There are cases where no existing plot provides a sensible starting point for creating the final plot that the user requires. This section describes how to construct a new plot entirely from scratch for such cases.

The `plot.new()` function is the most basic starting point for producing a traditional graphics plot (the `frame()` function is equivalent). This function sets up the various plotting regions described in Section 3.1.1 and sets both the x-scale and y-scale to $(0, 1)$.* The regions that are set up depend on the

*The actual scale setup depends on the current settings for `xaxs` and `yaxs`. With the default settings, the scales are $(-0.04, 1.04)$.

current graphics state settings.

The `plot.window()` function resets the scales in the user coordinate system, given x- and y-ranges via the arguments `xlim` and `ylim`, and the `plot.xy()` function draws data symbols and lines between locations within the plot region.

3.5.1 A simple plot from scratch

In order to demonstrate the use of these functions, the following code produces the simple scatterplot in Figure 1.1 from scratch.

```
> plot.new()
> plot.window(range(pressure$temperature),
              range(pressure$pressure))
> plot.xy(pressure, type="p")
> box()
> axis(1)
> axis(2)
```

The output could be produced by the simple expression `plot(pressure)`, but it shows that the steps in building a plot are available as separate functions as well, which allows the user to have fine control over the construction of a plot.

3.5.2 A more complex plot from scratch

This section describes a slightly more complex example of creating a plot from scratch. The final goal is represented in Figure 3.30 and the steps involved are described below.

This first bit of code generates some data to plot.

```
> groups <- c("cows", "sheep", "horses",
              "elephants", "giraffes")
> males <- sample(1:10, 5)
> females <- sample(1:10, 5)
```

There are several ways that the plot could be created. For this example, it will be created as a single plot. The labels to the left of the plot will be drawn in the margins of the plot, but everything else will be drawn inside the plot region. This next bit of code sets up the figure margins so that there is enough

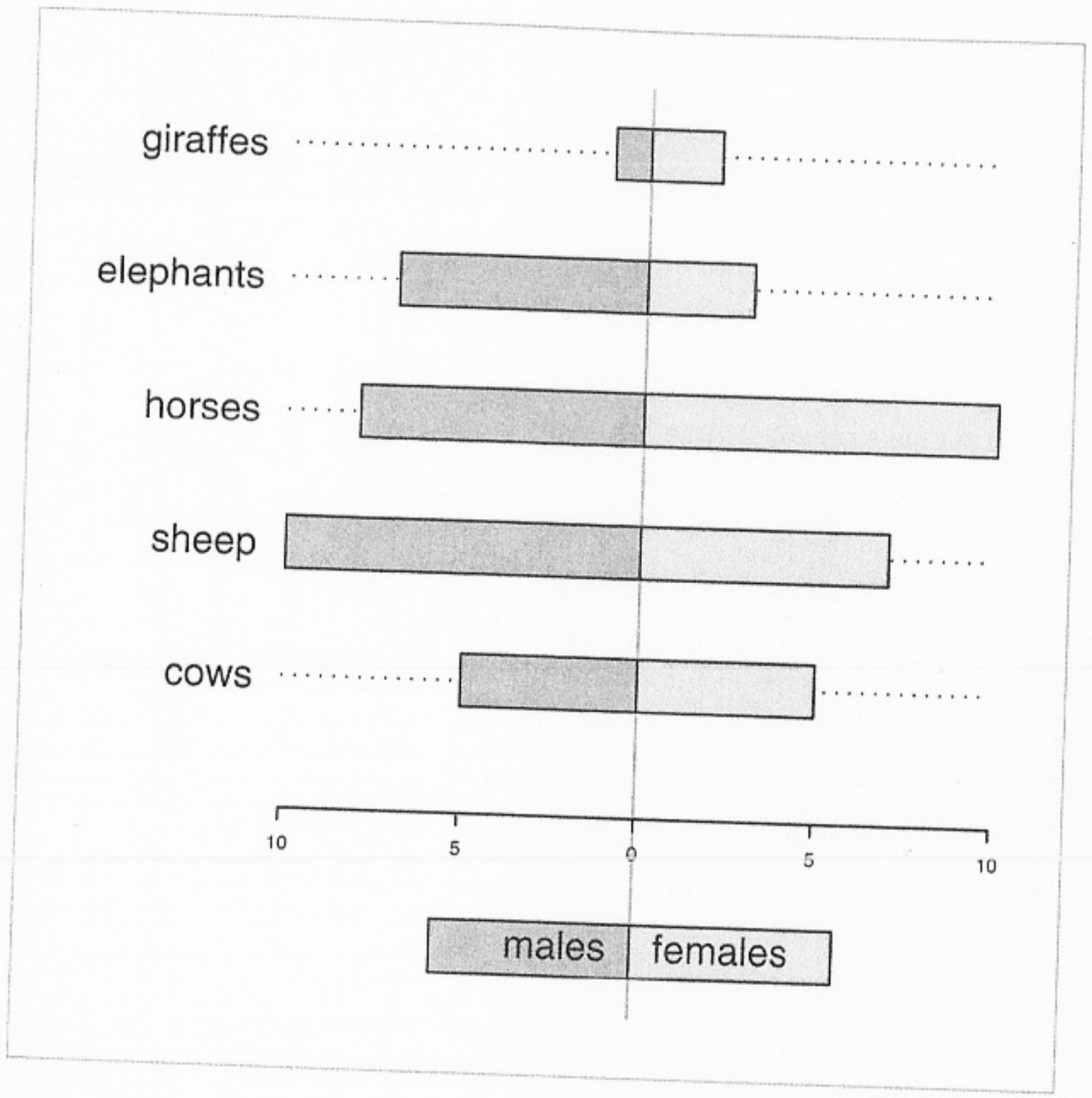


Figure 3.30
A back-to-back barplot from scratch. This demonstrates the use of lower-level plotting functions to produce a novel plot that cannot be produced by an existing high-level function.

room for the labels in the left margin, but all other margins are nice and small (to avoid lots of empty space around the plot).

```
> par(mar=c(0.5, 5, 0.5, 1))
```

Inside the plot region there are seven different rows of output to draw: the five main pairs of bars, the x-axis, and the legend at the bottom. The axis will be drawn at a y-location of `0`, the main bars at the y-locations `1:5`, and the legend at `-1`. The following code starts the plot and sets up the appropriate y-scale and x-scale.

```
> plot.new()
> plot.window(xlim=c(-10, 10), ylim=c(-1.5, 5.5))
```

This next bit of code assigns some useful values to variables, including the x-locations of tick-marks on the x-axis, the y-locations of the main bars, and a value representing half the height of the bars.

```
> ticks <- seq(-10, 10, 5)
> y <- 1:5
> h <- 0.2
```

Now some drawing can occur. This next code draws the main part of the plot. Everything is drawn using calls to the low-level functions such as `lines()`, `segments()`, `mtext()`, and `axis()`. In particular, the main bars are just rectangles produced using `rect()`. Notice that the x-axis is drawn within the plot region (`pos=0`).

```
> lines(rep(0, 2), c(-1.5, 5.5), col="grey")
> segments(-10, y, 10, y, lty="dotted")
> rect(-males, y-h, 0, y+h, col="dark grey")
> rect(0, y-h, females, y+h, col="light grey")
> mtext(groups, at=y, adj=1, side=2, las=2)
> par(cex.axis=0.5, mex=0.5)
> axis(1, at=ticks, labels=abs(ticks), pos=0)
```

The final step is to produce the legend at the bottom of the plot. Again, this is just a series of calls to low-level functions, although the bars are sized using `strwidth()` to ensure that they contain the labels.

```
> tw <- 1.5*strwidth("females")
> rect(-tw, -1-h, 0, -1+h, col="dark grey")
> rect(0, -1-h, tw, -1+h, col="light grey")
> text(0, -1, "males", pos=2)
> text(0, -1, "females", pos=4)
```

This example is particularly customized to the data set involved. It could be made much more general by replacing some constants with variable values (e.g., instead of using 5 because there are five groups in the data set, the code could have a variable `numGroups`). If more than one such plot needs to be made, it makes good sense to also wrap the code within a function. That task is discussed in the next section.

3.5.3 Writing traditional graphics functions

Having made the effort to construct a plot from scratch, it is usually worthwhile encapsulating the calls within a new function and possibly even making it available for others to use. This section briefly describes some of the things to consider when creating a new graphics function built on the traditional graphics system functions.

There are many advantages to developing new graphics functions in the grid graphics system (see Part II) rather than using traditional graphics. Chapter 7 contains a more complete discussion of the issues involved in developing new graphics functions.

Helper functions

There are some helper functions that do no drawing, but are used by the predefined high-level plots to do some of the work in setting up a plot.

The `xy.coords()` function is useful for allowing `x` and `y` arguments to your new function to be flexibly specified (just like the `plot()` function where `y` can be left unspecified and `x` can be a `data.frame`, and so on). This function takes `x` and `y` arguments and creates a standard object containing x-value, y-values, and sensible labels for the axes. There is also an `xyz.coords()` function.

If your plotting function generates multiple sub-plots, the `n2mfrow()` function may be helpful to generate a sensible number of rows and columns of plots, based on the total number of plots to fit on a page.

Another set of useful helper functions are those that calculate values to plot from the raw data (but do no actual drawing). Examples of these sorts of

functions are: `boxplot.stats()` used by `boxplot()` to generate five-number summaries; `contourLines()` used by `contour()` to generate contour lines; `nclass.Sturges()`, `nclass.scott()`, and `nclass.FD()` used by `hist()` to generate the number of intervals for a histogram; and `co.intervals()` used by `coplot()` to generate ranges of values for conditioning a data set into panels.

Some high-level functions invisibly return this sort of information too. For example, `boxplot()` returns the combined results from `boxplot.stats()` for all of the boxplots that it produces and `hist()` returns information on the intervals that it creates including the number of data values in each interval.

Argument lists

A common technique when writing a traditional graphics function is to provide an ellipsis argument (`...`) instead of individual graphics state arguments (such as `col` and `lty`). This allows users to specify any state settings (e.g., `col="red"` and `lty="dashed"`) and the new function can pass them straight on to the traditional graphics functions that the new function calls. This avoids having to specify all individual state settings as arguments to the new function. Some care must be taken with this technique because sometimes different graphics functions interpret the same graphics state setting in different ways (the `col` setting is a good example; see Section 3.2). In such cases, it becomes necessary to name the individual graphics state setting as an argument and explicitly pass it on only to other graphics calls that will accept it and respond to it in the desired manner.

Sometimes it is useful for a graphics function to deliberately override the current graphics state settings. For example, a new plot may want to force the `xpd` setting to be `NA` in order to draw lines and text outside of the plot region. In such cases, it is polite for the graphics function to revert the graphics state settings at the end of the function so that users do not get a nasty surprise! A standard technique is to put the following expressions at the start of the new function to restore the graphics state to the settings that existed before the function was called.

```
opar <- par(no.readonly=TRUE)
on.exit(par(opar))
```

Care should be taken to ensure that a new graphics function takes notice of appropriate graphics state settings (e.g., `ann`). This can be a little complicated to implement because it is necessary to be aware of the possibility that the user might specify a setting in the call to the function and that such a setting should override the main graphics state setting. The standard approach is

to name the state setting explicitly as an argument to the graphics function and provide the permanent state setting as a default value. See the new graphics function template below for an example of this technique using the ann argument. An additional complication is that now there is a state setting that will not be part of the ... argument, so the state setting must be explicitly passed on to any other functions that might make use of it.

Another good technique is to provide arguments that users are used to seeing in other graphics functions — the `main`, `sub`, `xlim`, and `ylim` arguments are good examples of this sort of thing — and a new graphics function should be able to handle missing and non-finite values. The functions `is.na()`, `is.finite()`, and `na.omit()` may be useful for this purpose.

Plot methods

If a new function is for use with a particular type of data, then it is convenient for users if the function is provided as a method for the generic `plot()` function. This allows users to simply call the new function by calling `plot(x)`, where `x` is an object of the relevant class.

A graphics function template

The code in Figure 3.31 is a simple shell that combines some of the basic guidelines from this section. This is just a simplified version of the default `plot()` method. It is far from complete and will not gracefully accept all possible inputs (especially via the ... argument), but it could be used as the starting template for writing a new traditional graphics function.


```
plot.newclass <-
  function(x, y=NULL,
           main="", sub="",
           xlim=NULL, ylim=NULL,
           axes=TRUE, ann=par("ann"),
           col=par("col"),
           ...) {
  xy <- xy.coords(x, y)
  if (is.null(xlim))
    xlim <- range(xy$x[is.finite(xy$x)])
  if (is.null(ylim))
    ylim <- range(xy$y[is.finite(xy$y)])
  opar <- par(no.readonly=TRUE)
  on.exit(par(opar))
  plot.new()
  plot.window(xlim, ylim, ...)
  points(xy$x, xy$y, col=col, ...)
  if (axes) {
    axis(1)
    axis(2)
    box()
  }
  if (ann)
    title(main=main, sub=sub,
          xlab=xy$xlab, ylab=xy$ylab, ...)
}
```

Figure 3.31
A graphics function template. This code provides a starting point for producing a new graphics function for others to use.

Chapter summary

High-level traditional graphics functions produce complete plots and low-level traditional graphics functions add output to existing plots. There are low-level functions for producing simple output such as lines, rectangles, text, and polygons and also functions for producing more complex output such as axes and legends.

The traditional graphics system creates several regions for drawing the various components of a plot: a plot region for drawing data symbols and lines, figure margins for axes and labels, and so on. Each low-level graphics function produces output in a particular drawing region and most work in the plot region.

There is a traditional graphics system state that consists of settings to control the appearance of output and the arrangement of the drawing regions. There are settings for controlling color, fonts, line styles, data symbol style, and the style of axes. There are several mechanisms for arranging multiple plots on a single page.

It is straightforward to create a complete plot using only low-level graphics functions. This makes it possible to produce a completely new type of plot. It is also possible for the user to define an entirely new graphics function.

Part II

GRID GRAPHICS