3 Getting started with graphs

This chapter covers

- Creating and saving graphs
- Customizing symbols, lines, colors, and axes
- Annotating with text and titles
- Controlling a graph's dimensions
- Combining multiple graphs into one

On many occasions, I've presented clients with carefully crafted statistical results in the form of numbers and text, only to have their eyes glaze over while the chirping of crickets permeated the room. Yet those same clients had enthusiastic "Ah-ha!" moments when I presented the same information to them in the form of graphs. Many times I was able to see patterns in data or detect anomalies in data values by looking at graphs—patterns or anomalies that I completely missed when conducting more formal statistical analyses.

Human beings are remarkably adept at discerning relationships from visual representations. A well-crafted graph can help you make meaningful comparisons among thousands of pieces of information, extracting patterns not easily found through other methods. This is one reason why advances in the field of statistical graphics have had such a major impact on data analysis. Data analysts need to *look* at their data, and this is one area where R shines.

In this chapter, we'll review general methods for working with graphs. We'll start with how to create and save graphs. Then we'll look at how to modify the features that are found in any graph. These features include graph titles, axes, labels, colors, lines, symbols, and text annotations. Our focus will be on generic techniques that apply across graphs. (In later chapters, we'll focus on specific types of graphs.) Finally, we'll investigate ways to combine multiple graphs into one overall graph.

3.1 Working with graphs

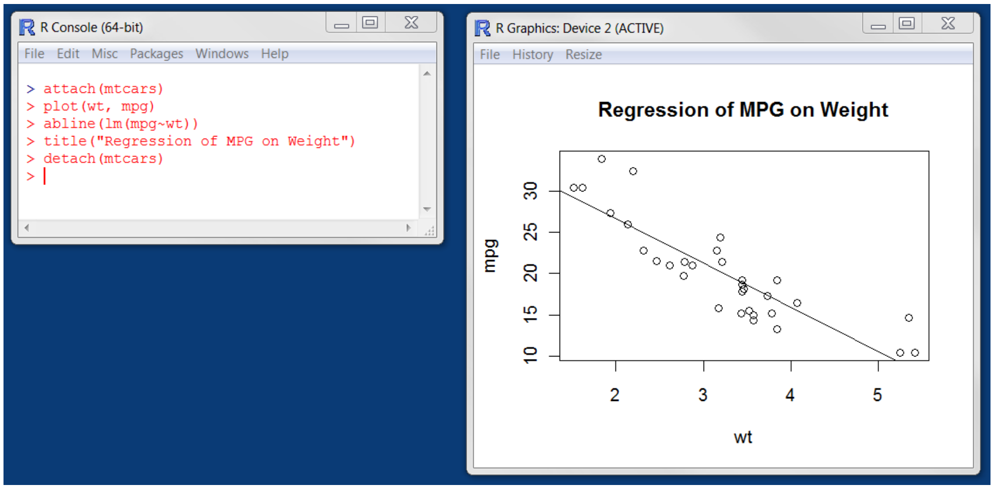
R is an amazing platform for building graphs. I'm using the term "building" intentionally. In a typical interactive session, you build a graph one statement at a time, adding features, until you have what you want.

Consider the following five lines:

```
attach(mtcars)
plot(wt, mpg)
abline(lm(mpg~wt))
title("Regression of MPG on Weight")
detach(mtcars)
```

The first statement attaches the data frame `mtcars`. The second statement opens a graphics window and generates a scatter plot between automobile weight on the horizontal axis and miles per gallon on the vertical axis. The third statement adds a line of best fit. The fourth statement adds a title. The final statement detaches the data frame. In R, graphs are typically created in this interactive fashion (see figure 3.1).

You can save your graphs via code or through GUI menus. To save a graph via code, sandwich the statements that produce the graph between a statement that sets a destination and a statement that closes that destination. For example, the following



```
> attach(mtcars)
> plot(wt, mpg)
> abline(lm(mpg~wt))
> title("Regression of MPG on Weight")
> detach(mtcars)
> |
```

R Console (64-bit)
File Edit Misc Packages Windows Help
R Graphics: Device 2 (ACTIVE)
File History Resize
Regression of MPG on Weight
mpg
10 15 20 25 30
2 3 4 5
wt

Figure 3.1 Creating a graph

will save the graph as a PDF document named `mygraph.pdf` in the current working directory:

```
pdf("mygraph.pdf")
 attach(mtcars)
 plot(wt, mpg)
 abline(lm(mpg~wt))
 title("Regression of MPG on Weight")
 detach(mtcars)
dev.off()
```

In addition to `pdf()`, you can use the functions `win.metafile()`, `png()`, `jpeg()`, `bmp()`, `tiff()`, `xfig()`, and `postscript()` to save graphs in other formats. (Note: The Windows metafile format is only available on Windows platforms.) See chapter 1, section 1.3.4 for more details on sending graphic output to files.

Saving graphs via the GUI will be platform specific. On a Windows platform, select File > Save As from the graphics window, and choose the format and location desired in the resulting dialog. On a Mac, choose File > Save As from the menu bar when the Quartz graphics window is highlighted. The only output format provided is PDF. On a Unix platform, the graphs must be saved via code. In appendix A, we'll consider alternative GUIs for each platform that will give you more options.

Creating a new graph by issuing a high-level plotting command such as `plot()`, `hist()` (for histograms), or `boxplot()` will typically overwrite a previous graph. How can you create more than one graph and still have access to each? There are several methods.

First, you can open a new graph window *before* creating a new graph:

```
dev.new()
 statements to create graph 1
dev.new()
 statements to create a graph 2
etc.
```

Each new graph will appear in the most recently opened window.

Second, you can access multiple graphs via the GUI. On a Mac platform, you can step through the graphs at any time using Back and Forward on the Quartz menu. On a Windows platform, you must use a two-step process. After opening the *first* graph window, choose History > Recording. Then use the Previous and Next menu items to step through the graphs that are created.

Third and finally, you can use the functions `dev.new()`, `dev.next()`, `dev.prev()`, `dev.set()`, and `dev.off()` to have multiple graph windows open at one time and choose which output are sent to which windows. This approach works on any platform. See `help(dev.cur)` for details on this approach.

R will create attractive graphs with a minimum of input on our part. But you can also use graphical parameters to specify fonts, colors, line styles, axes, reference lines, and annotations. This flexibility allows for a wide degree of customization.

In this chapter, we'll start with a simple graph and explore the ways you can modify and enhance it to meet your needs. Then we'll look at more complex examples that illustrate additional customization methods. The focus will be on techniques that you can apply to a wide range of the graphs that you'll create in R. The methods discussed here will work on all the graphs described in this book, with the exception of those created with the `lattice` package in chapter 16. (The `lattice` package has its own methods for customizing a graph's appearance.) In other chapters, we'll explore each specific type of graph and discuss where and when they're most useful.

3.2 A simple example

Let's start with the simple fictitious dataset given in table 3.1. It describes patient response to two drugs at five dosage levels.

Table 3.1 Patient response to two drugs at five dosage levels

Dosage	Response to Drug A	Response to Drug B
20	16	15
30	20	18
40	27	25
45	40	31
60	60	40

You can input this data using this code:

```
dose  <- c(20, 30, 40, 45, 60)
drugA <- c(16, 20, 27, 40, 60)
drugB <- c(15, 18, 25, 31, 40)
```

A simple line graph relating dose to response for drug A can be created using

```
plot(dose, drugA, type="b")
```

`plot()` is a generic function that plots objects in R (its output will vary according to the type of object being plotted). In this case, `plot(x, y, type="b")` places `x` on the horizontal axis and `y` on the vertical axis, plots the (x, y) data points, and connects them with line segments. The option `type="b"` indicates that both points and lines should be plotted. Use `help(plot)` to view other options. The graph is displayed in figure 3.2.

Line plots are covered in detail in chapter 11. Now let's modify the appearance of this graph.

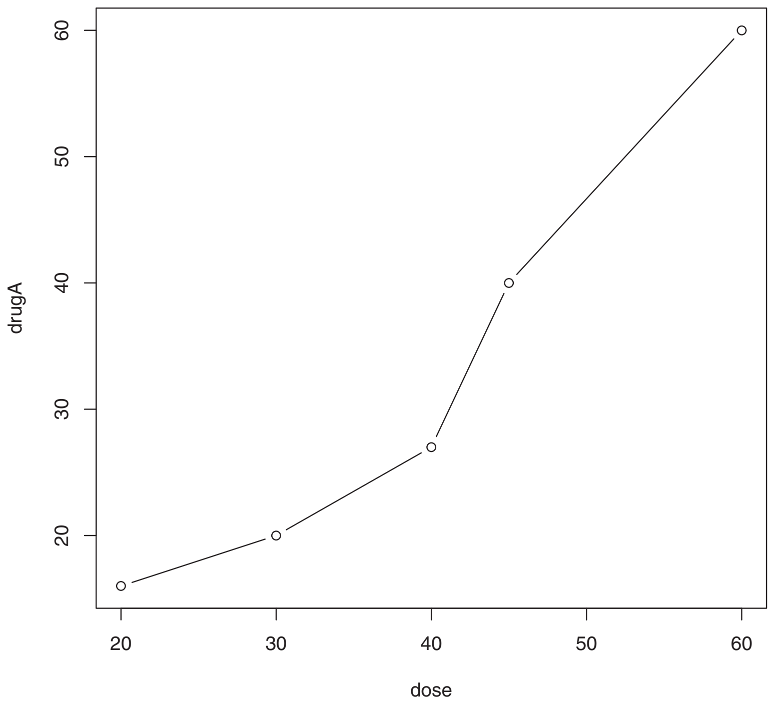


Figure 3.2 **Line plot of dose vs. response for drug A**

3.3 Graphical parameters

You can customize many features of a graph (fonts, colors, axes, titles) through options called *graphical parameters*.

One way is to specify these options through the `par()` function. Values set in this manner will be in effect for the rest of the session or until they're changed. The format is `par(optionname=value, optionname=value, ...)`. Specifying `par()` without parameters produces a list of the current graphical settings. Adding the `no.readonly=TRUE` option produces a list of current graphical settings that can be modified.

Continuing our example, let's say that you'd like to use a solid triangle rather than an open circle as your plotting symbol, and connect points using a dashed line rather than a solid line. You can do so with the following code:

```
opar <- par(no.readonly=TRUE)
par(lty=2, pch=17)
plot(dose, drugA, type="b")
par(opar)
```

The resulting graph is shown in figure 3.3.

The first statement makes a copy of the current settings. The second statement changes the default line type to dashed (`lty=2`) and the default symbol for plotting

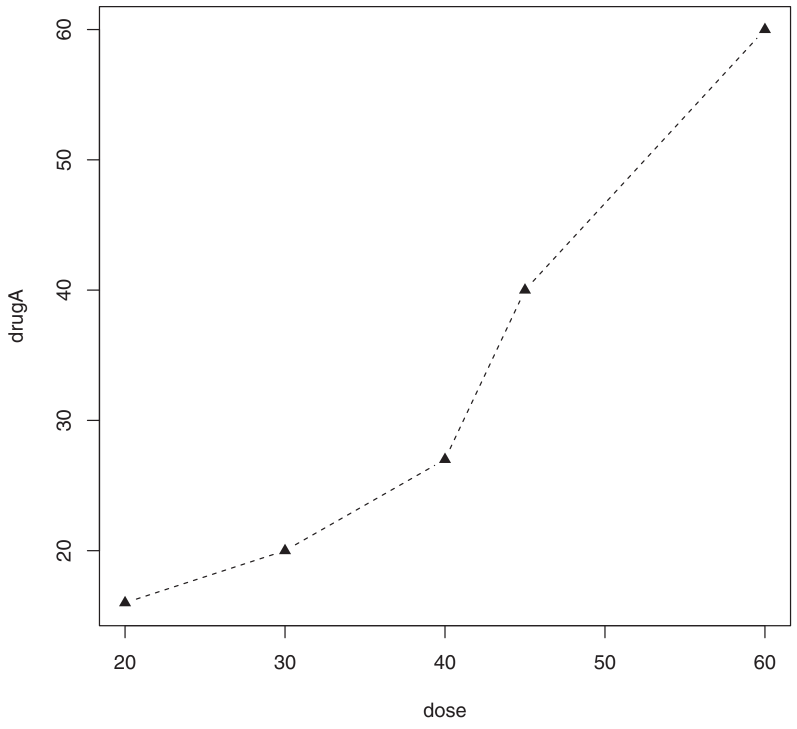


Figure 3.3 **Line plot of dose vs. response for drug A with modified line type and symbol**

points to a solid triangle (`pch=17`). You then generate the plot and restore the original settings. Line types and symbols are covered in section 3.3.1.

You can have as many `par()` functions as desired, so `par(lty=2, pch=17)` could also have been written as

```
par(lty=2)
par(pch=17)
```

A second way to specify graphical parameters is by providing the *`optionname=value`* pairs directly to a high-level plotting function. In this case, the options are only in effect for that specific graph. You could've generated the same graph with the code

```
plot(dose, drugA, type="b", lty=2, pch=17)
```

Not all high-level plotting functions allow you to specify all possible graphical parameters. See the help for a specific plotting function (such as `?plot`, `?hist`, or `?boxplot`) to determine which graphical parameters can be set in this way. The remainder of section 3.3 describes many of the important graphical parameters that you can set.

3.3.1 Symbols and lines

As you've seen, you can use graphical parameters to specify the plotting symbols and lines used in your graphs. The relevant parameters are shown in table 3.2.

Table 3.2 Parameters for specifying symbols and lines

Parameter	Description
`pch`	Specifies the symbol to use when plotting points (see figure 3.4).
`cex`	Specifies the symbol size. `cex` is a number indicating the amount by which plotting symbols should be scaled relative to the default. 1=default, 1.5 is 50% larger, 0.5 is 50% smaller, and so forth.
`lty`	Specifies the line type (see figure 3.5).
`lwd`	Specifies the line width. `lwd` is expressed relative to the default (default=1). For example, `lwd=2` generates a line twice as wide as the default.

The `pch=` option specifies the symbols to use when plotting points. Possible values are shown in figure 3.4.

For symbols 21 through 25 you can also specify the border (`col=`) and fill (`bg=`) colors.

Use `lty=` to specify the type of line desired. The option values are shown in figure 3.5.

Taking these options together, the code

```
plot(dose, drugA, type="b", lty=3, lwd=3, pch=15, cex=2)
```

would produce a plot with a dotted line that was three times wider than the default width, connecting points displayed as filled squares that are twice as large as the default symbol size. The results are displayed in figure 3.6.

Next, let's look at specifying colors.

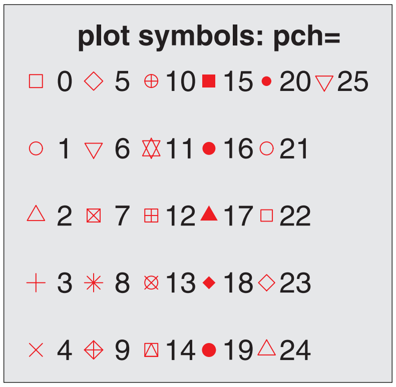


Figure 3.4 Plotting symbols specified with the `pch` parameter

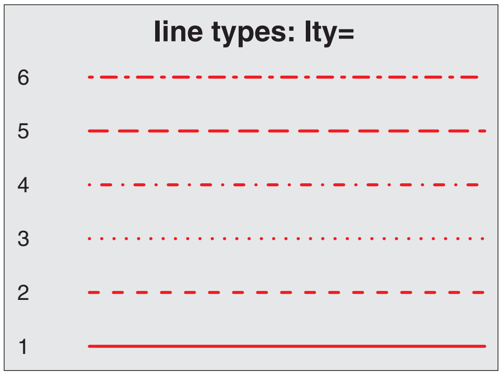


Figure 3.5 Line types specified with the `lty` parameter

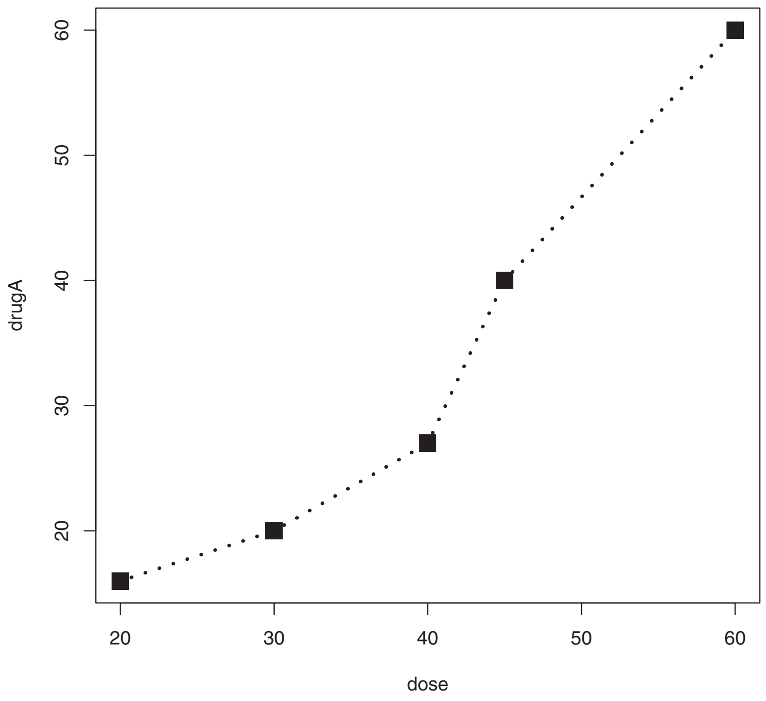


Figure 3.6 Line plot of dose vs. response for drug A with modified line type, line width, symbol, and symbol width

3.3.2 Colors

There are several color-related parameters in R. Table 3.3 shows some of the common ones.

Table 3.3 Parameters for specifying color

Parameter	Description
`col`	Default plotting color. Some functions (such as lines and pie) accept a vector of values that are recycled. For example, if `col=c("red", "blue")` and three lines are plotted, the first line will be red, the second blue, and the third red.
`col.axis`	Color for axis text.
`col.lab`	Color for axis labels.
`col.main`	Color for titles.
`col.sub`	Color for subtitles.
`fg`	The plot's foreground color.
`bg`	The plot's background color.

You can specify colors in R by index, name, hexadecimal, RGB, or HSV. For example, `col=1`, `col="white"`, `col="#FFFFFF"`, `col=rgb(1,1,1)`, and `col=hsv(0,0,1)` are equivalent ways of specifying the color white. The function `rgb()` creates colors based on red-green-blue values, whereas `hsv()` creates colors based on hue-saturation values. See the help feature on these functions for more details.

The function `colors()` returns all available color names. Earl F. Glynn has created an excellent online chart of R colors, available at http://research.stowers-institute.org/efg/R/Color/Chart. R also has a number of functions that can be used to create vectors of contiguous colors. These include `rainbow()`, `heat.colors()`, `terrain.colors()`, `topo.colors()`, and `cm.colors()`. For example, `rainbow(10)` produces 10 contiguous "rainbow" colors. Gray levels are generated with the `gray()` function. In this case, you specify gray levels as a vector of numbers between 0 and 1. `gray(0:10/10)` would produce 10 gray levels. Try the code

```
n <- 10
mycolors <- rainbow(n)
pie(rep(1, n), labels=mycolors, col=mycolors)
mygrays <- gray(0:n/n)
pie(rep(1, n), labels=mygrays, col=mygrays)
```

to see how this works. You'll see examples that use color parameters throughout this chapter.

3.3.3 *Text characteristics*

Graphic parameters are also used to specify text size, font, and style. Parameters controlling text size are explained in table 3.4. Font family and style can be controlled with font options (see table 3.5).

Table 3.4 Parameters specifying text size

Parameter	Description
`cex`	Number indicating the amount by which plotted text should be scaled relative to the default. 1=default, 1.5 is 50% larger, 0.5 is 50% smaller, etc.
`cex.axis`	Magnification of axis text relative to `cex`.
`cex.lab`	Magnification of axis labels relative to `cex`.
`cex.main`	Magnification of titles relative to `cex`.
`cex.sub`	Magnification of subtitles relative to `cex`.

For example, all graphs created after the statement

```
par(font.lab=3, cex.lab=1.5, font.main=4, cex.main=2)
```

will have italic axis labels that are 1.5 times the default text size, and bold italic titles that are twice the default text size.

Table 3.5 **Parameters specifying font family, size, and style**

Parameter	Description
`font`	Integer specifying font to use for plotted text.. 1=plain, 2=bold, 3=italic, 4=bold italic, 5=symbol (in Adobe symbol encoding).
`font.axis`	Font for axis text.
`font.lab`	Font for axis labels.
`font.main`	Font for titles.
`font.sub`	Font for subtitles.
`ps`	Font point size (roughly 1/72 inch). The text size = `ps*cex`.
`family`	Font family for drawing text. Standard values are `serif`, `sans`, and `mono`.

Whereas font size and style are easily set, font family is a bit more complicated. This is because the mapping of serif, sans, and mono are device dependent. For example, on Windows platforms, mono is mapped to TT Courier New, serif is mapped to TT Times New Roman, and sans is mapped to TT Arial (TT stands for True Type). If you're satisfied with this mapping, you can use parameters like `family="serif"` to get the results you want. If not, you need to create a new mapping. On Windows, you can create this mapping via the `windowsFont()` function. For example, after issuing the statement

```
windowsFonts(
  A=windowsFont("Arial Black"),
  B=windowsFont("Bookman Old Style"),
  C=windowsFont("Comic Sans MS")
)
```

you can use A, B, and C as family values. In this case, `par(family="A")` will specify an Arial Black font. (Listing 3.2 in section 3.4.2 provides an example of modifying text parameters.) Note that the `windowsFont()` function only works for Windows. On a Mac, use `quartzFonts()` instead.

If graphs will be output in PDF or PostScript format, changing the font family is relatively straightforward. For PDFs, use `names(pdfFonts())`to find out which fonts are available on your system and `pdf(file="`*`myplot`*`.pdf", family="`*`fontname`*`")` to generate the plots. For graphs that are output in PostScript format, use `names(postscriptFonts())` and `postscript(file="`*`myplot`*`.ps", family="`*`fontname`*`")`. See the online help for more information.

3.3.4 *Graph and margin dimensions*

Finally, you can control the plot dimensions and margin sizes using the parameters listed in table 3.6.

Table 3.6 Parameters for graph and margin dimensions

Parameter	Description
`pin`	Plot dimensions (width, height) in inches.
`mai`	Numerical vector indicating margin size where c(bottom, left, top, right) is expressed in inches.
`mar`	Numerical vector indicating margin size where c(bottom, left, top, right) is expressed in lines. The default is c(5, 4, 4, 2) + 0.1.

The code

```
par(pin=c(4,3), mai=c(1,.5, 1, .2))
```

produces graphs that are 4 inches wide by 3 inches tall, with a 1-inch margin on the bottom and top, a 0.5-inch margin on the left, and a 0.2-inch margin on the right. For a complete tutorial on margins, see Earl F. Glynn's comprehensive online tutorial (http://research.stowers-institute.org/efg/R/Graphics/Basics/mar-oma/).

Let's use the options we've covered so far to enhance our simple example. The code in the following listing produces the graphs in figure 3.7.

Listing 3.1 Using graphical parameters to control graph appearance

```
dose  <- c(20, 30, 40, 45, 60)
drugA <- c(16, 20, 27, 40, 60)
drugB <- c(15, 18, 25, 31, 40)
opar <- par(no.readonly=TRUE)
par(pin=c(2, 3))
par(lwd=2, cex=1.5)
par(cex.axis=.75, font.axis=3)
plot(dose, drugA, type="b", pch=19, lty=2, col="red")
plot(dose, drugB, type="b", pch=23, lty=6, col="blue", bg="green")
par(opar)
```

First you enter your data as vectors, then save the current graphical parameter settings (so that you can restore them later). You modify the default graphical parameters so that graphs will be 2 inches wide by 3 inches tall. Additionally, lines will be twice the default width and symbols will be 1.5 times the default size. Axis text will be set to italic and scaled to 75 percent of the default. The first plot is then created using filled red circles and dashed lines. The second plot is created using filled green filled diamonds and a blue border and blue dashed lines. Finally, you restore the original graphical parameter settings.

Note that parameters set with the `par()` function apply to both graphs, whereas parameters specified in the plot functions only apply to that specific graph. Looking at figure 3.7 you can see some limitations in your presentation. The graphs lack titles and the vertical axes are not on the same scale, limiting your ability to compare the two drugs directly. The axis labels could also be more informative.

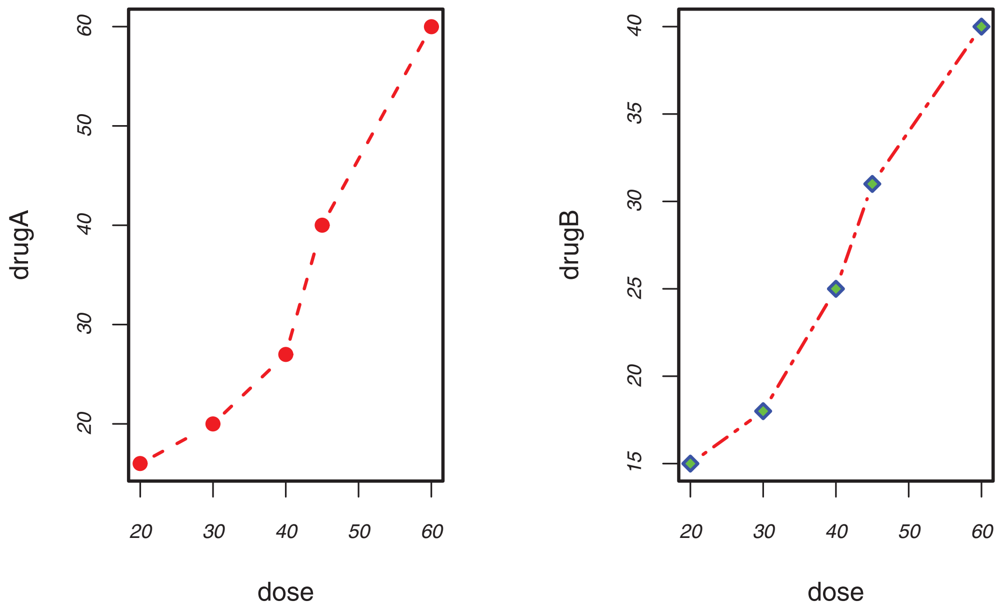


Figure 3.7 **Line plot of dose vs. response for both drug A and drug B**

In the next section, we'll turn to the customization of text annotations (such as titles and labels) and axes. For more information on the graphical parameters that are available, take a look at `help(par)`.

3.4 Adding text, customized axes, and legends

Many high-level plotting functions (for example, `plot`, `hist`, `boxplot`) allow you to include axis and text options, as well as graphical parameters. For example, the following adds a title (`main`), subtitle (`sub`), axis labels (`xlab`, `ylab`), and axis ranges (`xlim`, `ylim`). The results are presented in figure 3.8:

```
plot(dose, drugA, type="b",
     col="red", lty=2, pch=2, lwd=2,
     main="Clinical Trials for Drug A",
     sub="This is hypothetical data",
     xlab="Dosage", ylab="Drug Response",
     xlim=c(0, 60), ylim=c(0, 70))
```

Again, not all functions allow you to add these options. See the help for the function of interest to see what options are accepted. For finer control and for modularization, you can use the functions described in the remainder of this section to control titles, axes, legends, and text annotations.

> **NOTE** Some high-level plotting functions include default titles and labels. You can remove them by adding `ann=FALSE` in the `plot()` statement or in a separate `par()` statement.

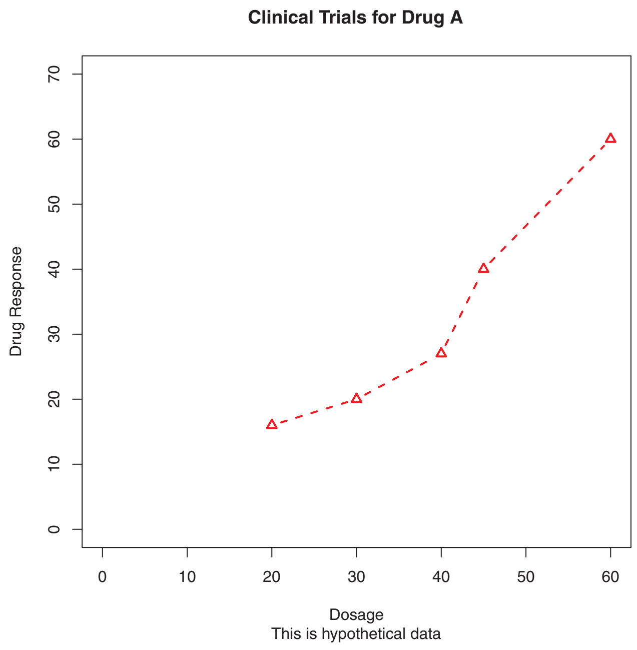


Figure 3.8 Line plot of dose versus response for drug A with title, subtitle, and modified axes

3.4.1 Titles

Use the `title()` function to add title and axis labels to a plot. The format is

```
title(main="main title", sub="sub-title",
      xlab="x-axis label", ylab="y-axis label")
```

Graphical parameters (such as text size, font, rotation, and color) can also be specified in the `title()` function. For example, the following produces a red title and a blue subtitle, and creates green x and y labels that are 25 percent smaller than the default text size:

```
title(main="My Title", col.main="red",
      sub="My Sub-title", col.sub="blue",
      xlab="My X label", ylab="My Y label",
      col.lab="green", cex.lab=0.75)
```

3.4.2 Axes

Rather than using R's default axes, you can create custom axes with the `axis()` function. The format is

```
axis(side, at=, labels=, pos=, lty=, col=, las=, tck=, ...)
```

where each parameter is described in table 3.7.

When creating a custom axis, you should suppress the axis automatically generated by the high-level plotting function. The option `axes=FALSE` suppresses all axes (including all axis frame lines, unless you add the option `frame.plot=TRUE`). The options `xaxt="n"` and `yaxt="n"` suppress the x- and y-axis, respectively (leaving the frame

Table 3.7 **Axis options**

Option	Description
`side`	An integer indicating the side of the graph to draw the axis (1=bottom, 2=left, 3=top, 4=right).
`at`	A numeric vector indicating where tick marks should be drawn.
`labels`	A character vector of labels to be placed at the tick marks (if `NULL`, the `at` values will be used).
`pos`	The coordinate at which the axis line is to be drawn (that is, the value on the other axis where it crosses).
`lty`	Line type.
`col`	The line and tick mark color.
`las`	Labels are parallel (`=0`) or perpendicular (`=2`) to the axis.
`tck`	Length of tick mark as a fraction of the plotting region (a negative number is outside the graph, a positive number is inside, 0 suppresses ticks, 1 creates gridlines); the default is –0.01.
`(...)`	Other graphical parameters.

lines, without ticks). The following listing is a somewhat silly and overblown example that demonstrates each of the features we've discussed so far. The resulting graph is presented in figure 3.9.

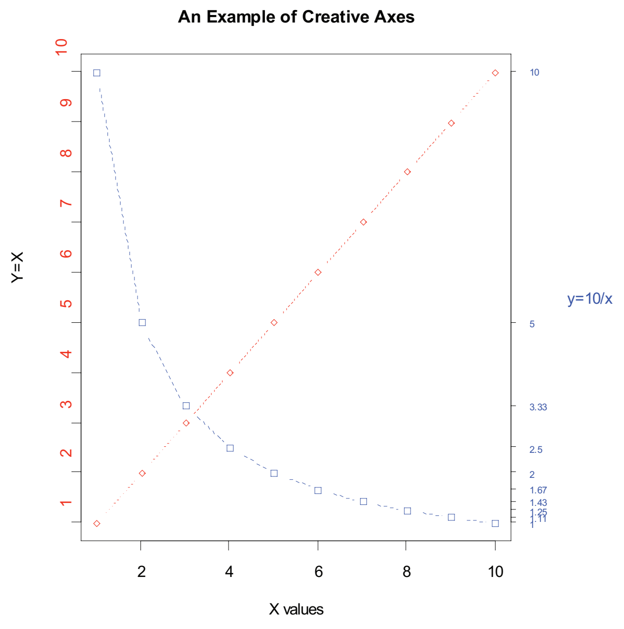


Figure 3.9 A demonstration of axis options

Listing 3.2 An example of custom axes

```
x <- c(1:10)                                 ◁— Specify data
y <- x
z <- 10/x

opar <- par(no.readonly=TRUE)

par(mar=c(5, 4, 4, 8) + 0.1)                 ◁— Increase margins

plot(x, y, type="b",                         ◁— Plot x versus y
     pch=21, col="red",
     yaxt="n", lty=3, ann=FALSE)
                                                Add x versus
lines(x, z, type="b", pch=22, col="blue", lty=2)   ◁— 1/x line

axis(2, at=x, labels=x, col.axis="red", las=2)  ◁— Draw your axes

axis(4, at=z, labels=round(z, digits=2),
     col.axis="blue", las=2, cex.axis=0.7, tck=-.01)

mtext("y=1/x", side=4, line=3, cex.lab=1, las=2, col="blue")  ◁— Add titles
                                                                  and text
title("An Example of Creative Axes",
      xlab="X values",
      ylab="Y=X")

par(opar)
```

At this point, we've covered everything in listing 3.2 except for the `line()` and the `mtext()` statements. A `plot()` statement starts a new graph. By using the `line()` statement instead, you can add new graph elements to an *existing* graph. You'll use it again when you plot the response of drug A and drug B on the same graph in section 3.4.4. The `mtext()` function is used to add text to the margins of the plot. The `mtext()` function is covered in section 3.4.5, and the `line()` function is covered more fully in chapter 11.

MINOR TICK MARKS

Notice that each of the graphs you've created so far have major tick marks but not minor tick marks. To create minor tick marks, you'll need the `minor.tick()` function in the `Hmisc` package. If you don't already have `Hmisc` installed, be sure to install it first (see chapter 1, section 1.4.2). You can add minor tick marks with the code

```
library(Hmisc)
minor.tick(nx=n, ny=n, tick.ratio=n)
```

where `nx` and `ny` specify the number of intervals in which to divide the area between major tick marks on the x-axis and y-axis, respectively. `tick.ratio` is the size of the minor tick mark relative to the major tick mark. The current length of the major tick mark can be retrieved using `par("tck")`. For example, the following statement will add one tick mark between each major tick mark on the x-axis and two tick marks between each major tick mark on the y-axis:

```
minor.tick(nx=2, ny=3, tick.ratio=0.5)
```

The length of the tick marks will be 50 percent as long as the major tick marks. An example of minor tick marks is given in the next section (listing 3.3 and figure 3.10).

3.4.3 Reference lines

The `abline()` function is used to add reference lines to our graph. The format is

```
abline(h=yvalues, v=xvalues)
```

Other graphical parameters (such as line type, color, and width) can also be specified in the `abline()` function. For example:

```
abline(h=c(1,5,7))
```

adds solid horizontal lines at y = 1, 5, and 7, whereas the code

```
abline(v=seq(1, 10, 2), lty=2, col="blue")
```

adds dashed blue vertical lines at x = 1, 3, 5, 7, and 9. Listing 3.3 creates a reference line for our drug example at y = 30. The resulting graph is displayed in figure 3.10.

3.4.4 Legend

When more than one set of data or group is incorporated into a graph, a legend can help you to identify what's being represented by each bar, pie slice, or line. A legend can be added (not surprisingly) with the `legend()` function. The format is

```
legend(location, title, legend, ...)
```

The common options are described in table 3.8.

Table 3.8 Legend options

Option	Description
`location`	There are several ways to indicate the location of the legend. You can give an x,y coordinate for the upper-left corner of the legend. You can use `locator(1)`, in which case you use the mouse to indicate the location of the legend. You can also use the keywords `bottom`, `bottomleft`, `left`, `topleft`, `top`, `topright`, `right`, `bottomright`, or `center` to place the legend in the graph. If you use one of these keywords, you can also use `inset=` to specify an amount to move the legend into the graph (as fraction of plot region).
`title`	A character string for the legend title (optional).
`legend`	A character vector with the labels.

Table 3.8 Legend options (*continued*)

Option	Description
`...`	Other options. If the legend labels colored lines, specify `col=` and a vector of colors. If the legend labels point symbols, specify `pch=` and a vector of point symbols. If the legend labels line width or line style, use `lwd=` or `lty=` and a vector of widths or styles. To create colored boxes for the legend (common in bar, box, or pie charts), use `fill=` and a vector of colors.

Other common legend options include `bty` for box type, `bg` for background color, `cex` for size, and `text.col` for text color. Specifying `horiz=TRUE` sets the legend horizontally rather than vertically. For more on legends, see `help(legend)`. The examples in the help file are particularly informative.

Let's take a look at an example using our drug data (listing 3.3). Again, you'll use a number of the features that we've covered up to this point. The resulting graph is presented in figure 3.10.

Listing 3.3 Comparing Drug A and Drug B response by dose

```
dose  <- c(20, 30, 40, 45, 60)
drugA <- c(16, 20, 27, 40, 60)
drugB <- c(15, 18, 25, 31, 40)


opar <- par(no.readonly=TRUE)


par(lwd=2, cex=1.5, font.lab=2)                    Increase line, text,
                                                   symbol, label size

plot(dose, drugA, type="b",                        Generate graph
     pch=15, lty=1, col="red", ylim=c(0, 60),
     main="Drug A vs. Drug B",
     xlab="Drug Dosage", ylab="Drug Response")

lines(dose, drugB, type="b",
      pch=17, lty=2, col="blue")

abline(h=c(30), lwd=1.5, lty=2, col="gray")

library(Hmisc)                                     Add minor tick
minor.tick(nx=3, ny=3, tick.ratio=0.5)             marks

legend("topleft", inset=.05, title="Drug Type", c("A","B"),   Add legend
       lty=c(1, 2), pch=c(15, 17), col=c("red", "blue"))

par(opar)
```

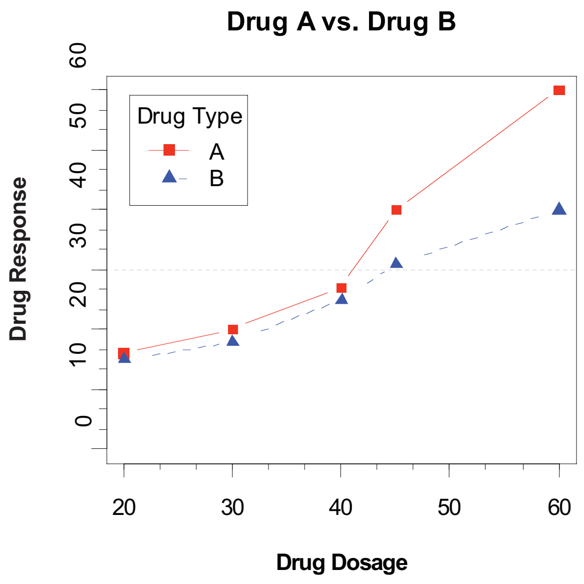


Figure 3.10 **An annotated comparison of Drug A and Drug B**

Almost all aspects of the graph in figure 3.10 can be modified using the options discussed in this chapter. Additionally, there are many ways to specify the options desired. The final annotation to consider is the addition of text to the plot itself. This topic is covered in the next section.

3.4.5 *Text annotations*

Text can be added to graphs using the `text()` and `mtext()` functions. `text()` places text within the graph whereas `mtext()` places text in one of the four margins. The formats are

```
text(location, "text to place", pos, ...)
mtext("text to place", side, line=n, ...)
```

and the common options are described in table 3.9.

Table 3.9 Options for the `text()` and `mtext()` functions

Option	Description
`location`	Location can be an x,y coordinate. Alternatively, the text can be placed interactively via mouse by specifying location as `locator(1)`.
`pos`	Position relative to location. 1 = below, 2 = left, 3 = above, 4 = right. If you specify `pos`, you can specify `offset=` in percent of character width.
`side`	Which margin to place text in, where 1 = bottom, 2 = left, 3 = top, 4 = right. You can specify `line=` to indicate the line in the margin starting with 0 (closest to the plot area) and moving out. You can also specify `adj=0` for left/bottom alignment or `adj=1` for top/right alignment.

Other common options are `cex`, `col`, and `font` (for size, color, and font style, respectively).

The `text()` function is typically used for labeling points as well as for adding other text annotations. Specify location as a set of x, y coordinates and specify the text to place as a vector of labels. The x, y, and label vectors should all be the same length. An example is given next and the resulting graph is shown in figure 3.11.

```
attach(mtcars)
plot(wt, mpg,
     main="Mileage vs. Car Weight",
     xlab="Weight", ylab="Mileage",
     pch=18, col="blue")
text(wt, mpg,
     row.names(mtcars),
     cex=0.6, pos=4, col="red")
detach(mtcars)
```

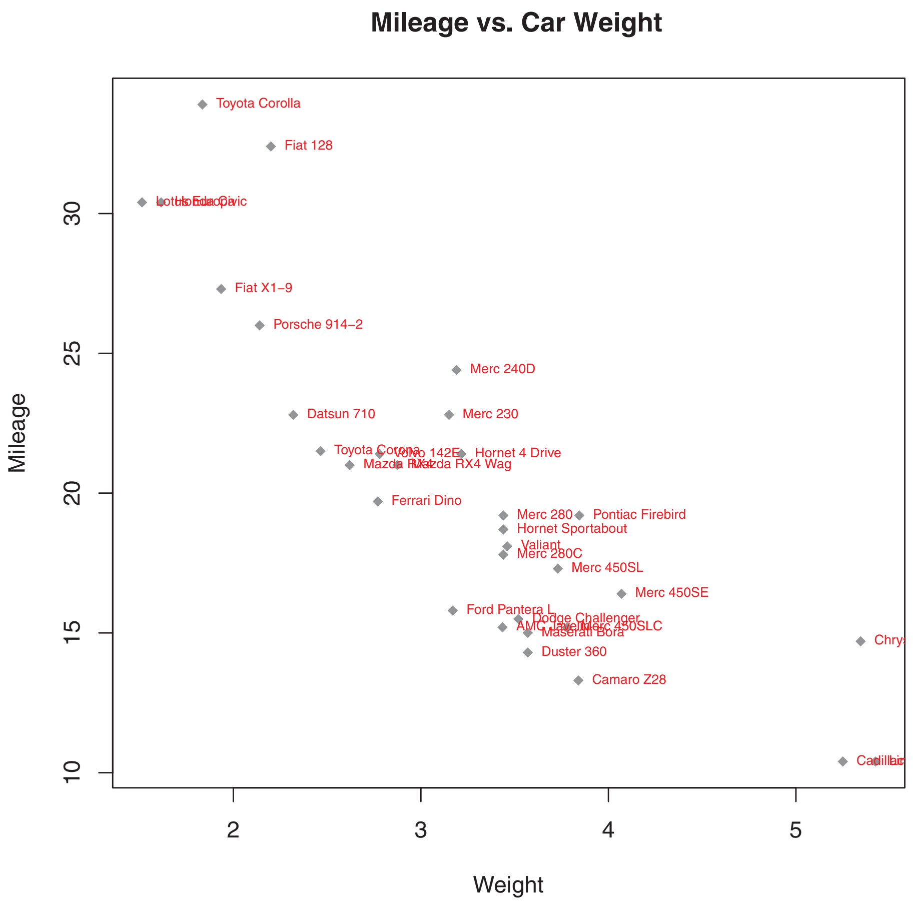


Figure 3.11 **Example of a scatter plot (car weight vs. mileage) with labeled points (car make)**

Here we've plotted car mileage versus car weight for the 32 automobile makes provided in the `mtcars` data frame. The `text()` function is used to add the car makes to the right of each data point. The point labels are shrunk by 40 percent and presented in red.

As a second example, the following code can be used to display font families:

```
opar <- par(no.readonly=TRUE)
par(cex=1.5)
plot(1:7,1:7,type="n")
text(3,3,"Example of default text")
text(4,4,family="mono","Example of mono-spaced text")
text(5,5,family="serif","Example of serif text")
par(opar)
```

The results, produced on a Windows platform, are shown in figure 3.12. Here the `par()` function was used to increase the font size to produce a better display.

The resulting plot will differ from platform to platform, because plain, mono, and serif text are mapped to different font families on different systems. What does it look like on yours?

MATH ANNOTATIONS

Finally, you can add mathematical symbols and formulas to a graph using TEX-like rules. See `help(plotmath)` for details and examples. You can also try `demo(plotmath)` to see this in action. A portion of the results is presented in figure 3.13. The `plotmath()` function can be used to add mathematical symbols to titles, axis labels, or text annotation in the body or margins of the graph.

You can often gain greater insight into your data by comparing several graphs at one time. So, we'll end this chapter by looking at ways to combine more than one graph into a single image.

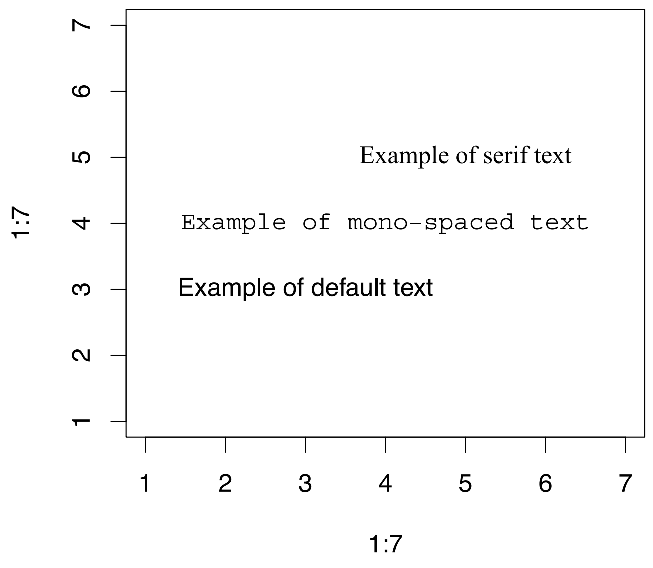
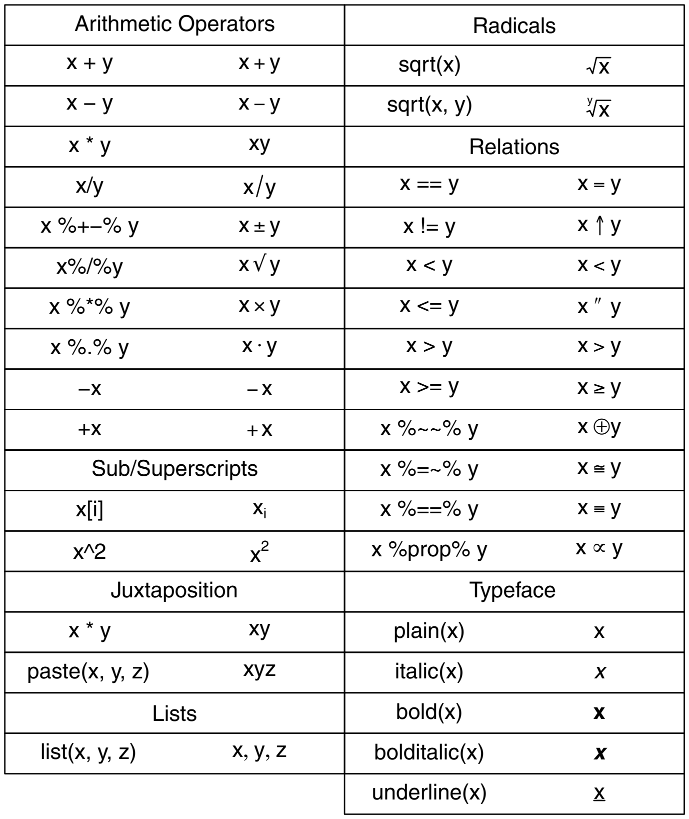


Figure 3.12 Examples of font families on a Windows platform



Arithmetic Operators	
x + y	x + y
x – y	x – y
x * y	xy
x/y	x/y
x %+–% y	x ± y
x%/%y	x √ y
x %*% y	x × y
x %.% y	x · y
–x	– x
+x	+ x
Sub/Superscripts	
x[i]	x_i
x^2	x^2
Juxtaposition	
x * y	xy
paste(x, y, z)	xyz
Lists	
list(x, y, z)	x, y, z

Radicals	
sqrt(x)	$\sqrt{x}$
sqrt(x, y)	$\sqrt[y]{x}$
Relations	
x == y	x = y
x != y	x ↑ y
x < y	x < y
x <= y	x ″ y
x > y	x > y
x >= y	x ≥ y
x %~~% y	x ⊕y
x %=~% y	x ≅ y
x %==% y	x ≡ y
x %prop% y	x ∝ y
Typeface	
plain(x)	x
italic(x)	*x*
bold(x)	**x**
bolditalic(x)	***x***
underline(x)	$\underline{x}$

Figure 3.13 **Partial results from `demo(plotmath)`**

3.5 Combining graphs

R makes it easy to combine several graphs into one overall graph, using either the `par()` or `layout()` function. At this point, don't worry about the specific types of graphs being combined; our focus here is on the general methods used to combine them. The creation and interpretation of each graph type is covered in later chapters.

With the `par()` function, you can include the graphical parameter `mfrow=c(`*`nrows`*`, `*`ncols`*`)` to create a matrix of *`nrows`* `x` *`ncols`* plots that are filled in by row. Alternatively, you can use `mfcol=c(`*`nrows`*`, `*`ncols`*`)` to fill the matrix by columns.

For example, the following code creates four plots and arranges them into two rows and two columns:

```
attach(mtcars)
opar <- par(no.readonly=TRUE)
par(mfrow=c(2,2))
plot(wt,mpg, main="Scatterplot of wt vs. mpg")
plot(wt,disp, main="Scatterplot of wt vs disp")
hist(wt, main="Histogram of wt")
```

```
boxplot(wt, main="Boxplot of wt")
par(opar)
detach(mtcars)
```

The results are presented in figure 3.14.

As a second example, let's arrange 3 plots in 3 rows and 1 column. Here's the code:

```
attach(mtcars)
opar <- par(no.readonly=TRUE)
par(mfrow=c(3,1))
hist(wt)
hist(mpg)
hist(disp)
par(opar)
detach(mtcars)
```

The graph is displayed in figure 3.15. Note that the high-level function `hist()` includes a default title (use `main=""` to suppress it, or `ann=FALSE` to suppress all titles and labels).

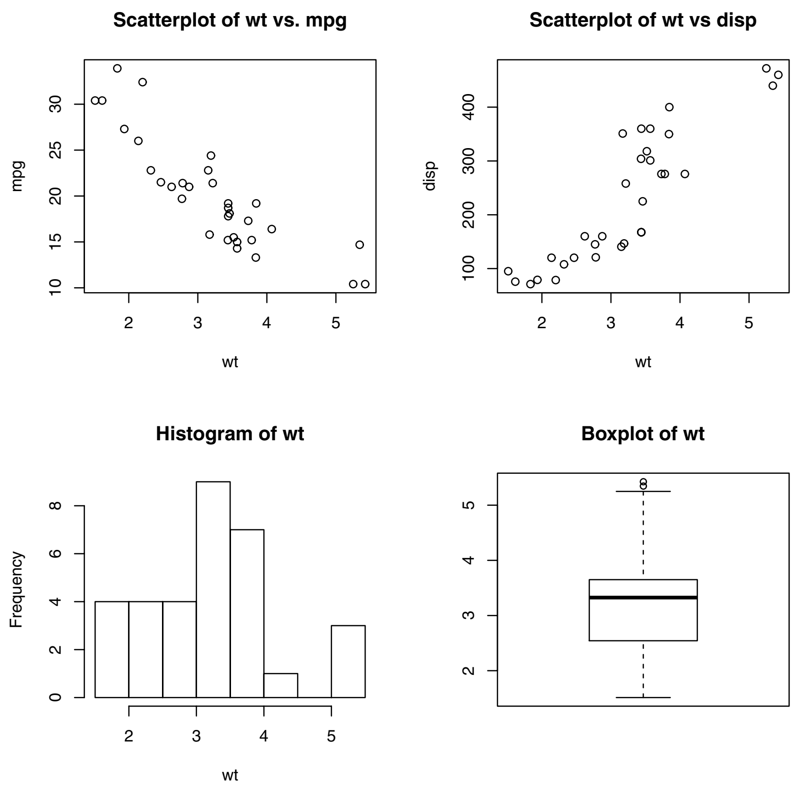


Figure 3.14 **Graph combining four figures through `par(mfrow=c(2,2))`**

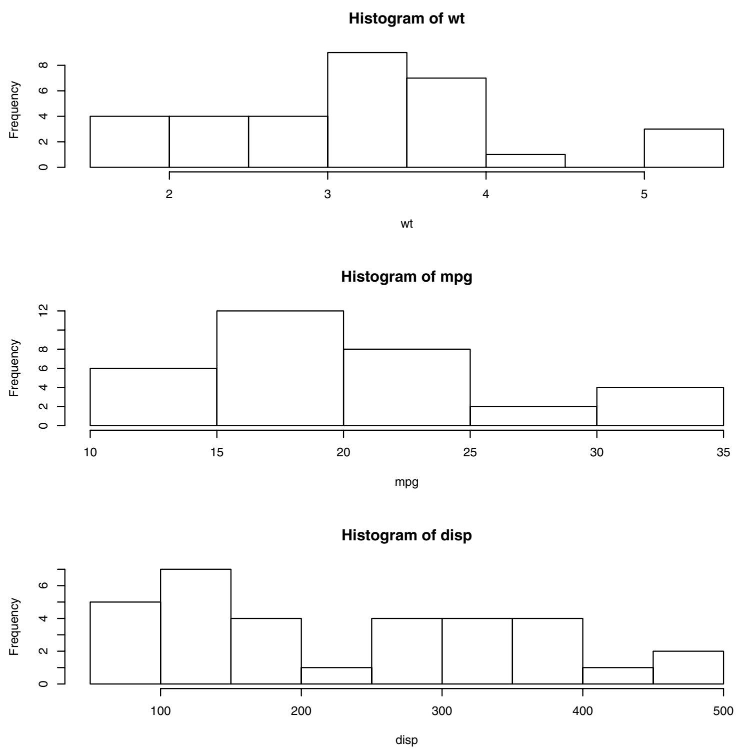


Figure 3.15 Graph combining with three figures through `par(mfrow=c(3,1))`

The `layout()` function has the form `layout(mat)` where *`mat`* is a matrix object specifying the location of the multiple plots to combine. In the following code, one figure is placed in row 1 and two figures are placed in row 2:

```
attach(mtcars)
layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE))
hist(wt)
hist(mpg)
hist(disp)
detach(mtcars)
```

The resulting graph is presented in figure 3.16.

Optionally, you can include `widths=` and `heights=` options in the `layout()` function to control the size of each figure more precisely. These options have the form

`widths` = a vector of values for the widths of columns
`heights` = a vector of values for the heights of rows

Relative widths are specified with numeric values. Absolute widths (in centimeters) are specified with the `lcm()` function.

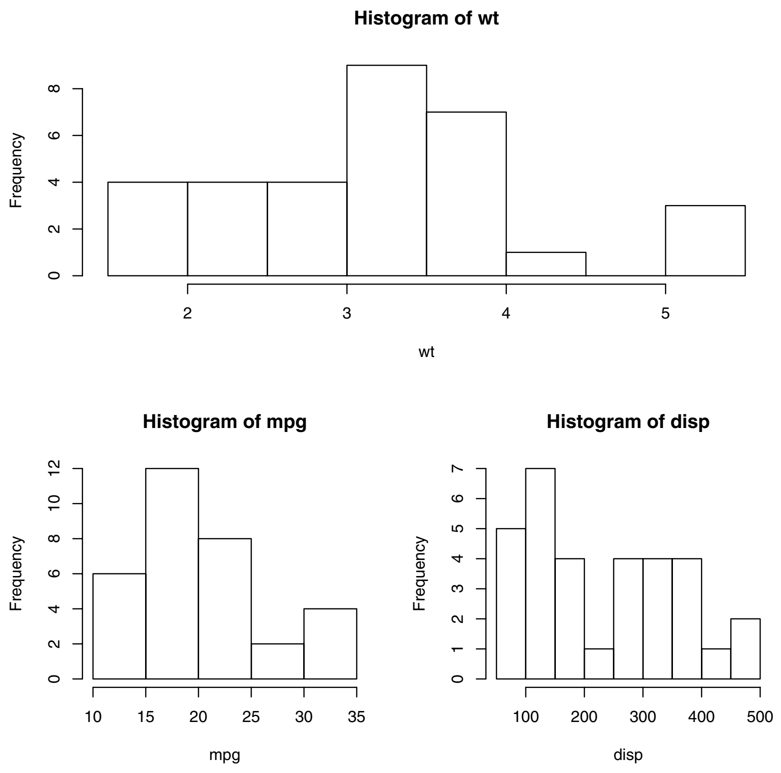


Figure 3.16 Graph combining three figures using the `layout()` function with default widths

In the following code, one figure is again placed in row 1 and two figures are placed in row 2. But the figure in row 1 is one-third the height of the figures in row 2. Additionally, the figure in the bottom-right cell is one-fourth the width of the figure in the bottom-left cell:

```
attach(mtcars)
layout(matrix(c(1, 1, 2, 3), 2, 2, byrow = TRUE),
       widths=c(3, 1), heights=c(1, 2))
hist(wt)
hist(mpg)
hist(disp)
detach(mtcars)
```

The graph is presented in figure 3.17.

As you can see, the `layout()` function gives you easy control over both the number and placement of graphs in a final image and the relative sizes of these graphs. See `help(layout)` for more details.

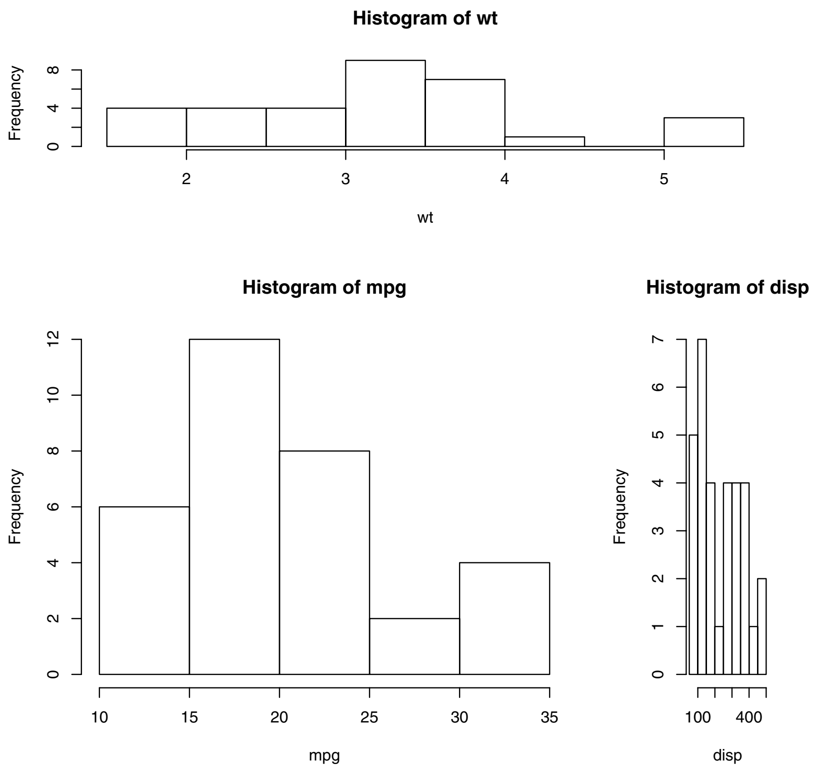


Figure 3.17 **Graph combining three figures using the `layout()` function with specified widths**

3.5.1 *Creating a figure arrangement with fine control*

There are times when you want to arrange or superimpose several figures to create a single meaningful plot. Doing so requires fine control over the placement of the figures. You can accomplish this with the `fig=` graphical parameter. In the following listing, two box plots are added to a scatter plot to create a single enhanced graph. The resulting graph is shown in figure 3.18.

Listing 3.4 Fine placement of figures in a graph

```
opar <- par(no.readonly=TRUE)
par(fig=c(0, 0.8, 0, 0.8))                                 <— Set up scatter plot
plot(mtcars$wt, mtcars$mpg,
     xlab="Miles Per Gallon",
     ylab="Car Weight")

par(fig=c(0, 0.8, 0.55, 1), new=TRUE)                      <— Add box plot above
boxplot(mtcars$wt, horizontal=TRUE, axes=FALSE)
```

```
par(fig=c(0.65, 1, 0, 0.8), new=TRUE)                    <- Add box plot to right
boxplot(mtcars$mpg, axes=FALSE)

mtext("Enhanced Scatterplot", side=3, outer=TRUE, line=-3)
par(opar)
```

To understand how this graph was created, think of the full graph area as going from (0,0) in the lower-left corner to (1,1) in the upper-right corner. Figure 3.19 will help you visualize this. The format of the `fig=` parameter is a numerical vector of the form `c(x1, x2, y1, y2)`.

The first `fig=` sets up the scatter plot going from 0 to 0.8 on the x-axis and 0 to 0.8 on the y-axis. The top box plot goes from 0 to 0.8 on the x-axis and 0.55 to 1 on the y-axis. The right-hand box plot goes from 0.65 to 1 on the x-axis and 0 to 0.8 on the y-axis. `fig=` starts a new plot, so when adding a figure to an existing graph, include the `new=TRUE` option.

I chose 0.55 rather than 0.8 so that the top figure would be pulled closer to the scatter plot. Similarly, I chose 0.65 to pull the right-hand box plot closer to the scatter plot. You have to experiment to get the placement right.

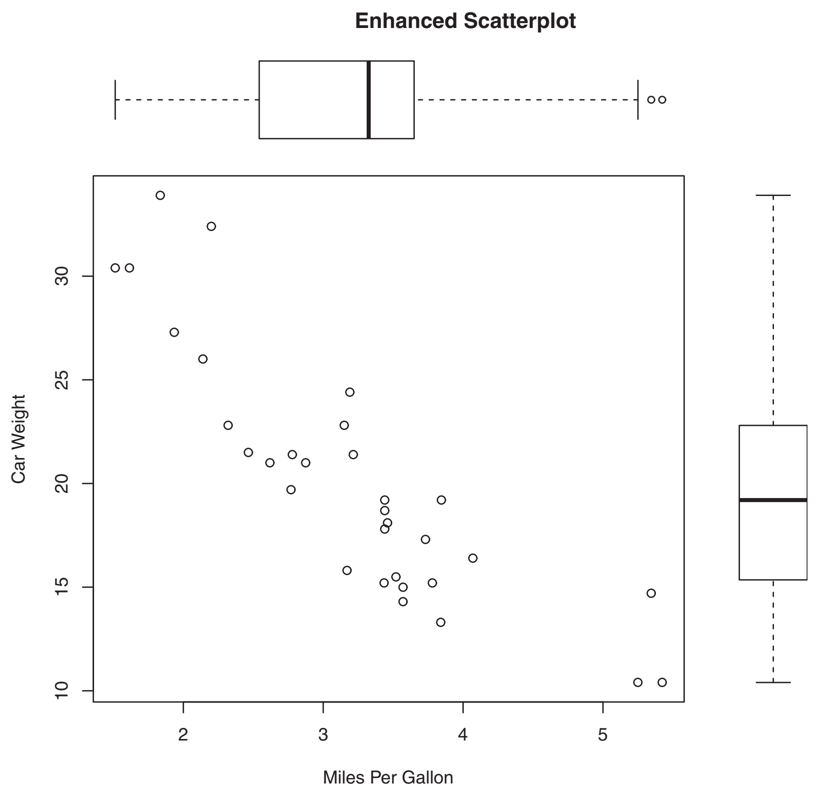


Figure 3.18 A scatter plot with two box plots added to the margins

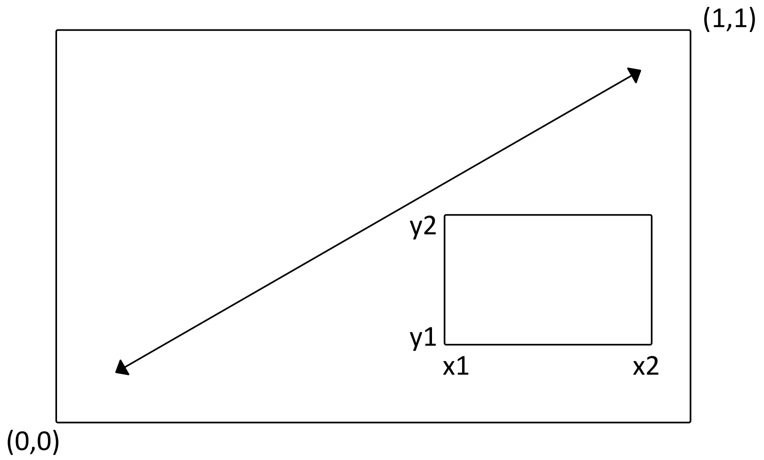


Figure 3.19 Specifying locations using the `fig=` graphical parameter

> **NOTE** The amount of space needed for individual subplots can be device dependent. If you get "Error in plot.new(): figure margins too large," try varying the area given for each portion of the overall graph.

You can use `fig=` graphical parameter to combine several plots into any arrangement within a single graph. With a little practice, this approach gives you a great deal of flexibility when creating complex visual presentations.

3.6 Summary

In this chapter, we reviewed methods for creating graphs and saving them in a variety of formats. The majority of the chapter was concerned with modifying the default graphs produced by R, in order to arrive at more useful or attractive plots. You learned how to modify a graph's axes, fonts, symbols, lines, and colors, as well as how to add titles, subtitles, labels, plotted text, legends, and reference lines. You saw how to specify the size of the graph and margins, and how to combine multiple graphs into a single useful image.

Our focus in this chapter was on general techniques that you can apply to all graphs (with the exception of lattice graphs in chapter 16). Later chapters look at specific types of graphs. For example, chapter 7 covers methods for graphing a single variable. Graphing relationships between variables will be described in chapter 11. In chapter 16, we discuss advanced graphic methods, including lattice graphs (graphs that display the relationship between variables, for each level of other variables) and interactive graphs. Interactive graphs let you use the mouse to dynamically explore the plotted relationships.

In other chapters, we'll discuss methods of visualizing data that are particularly useful for the statistical approaches under consideration. Graphs are a central part of modern data analysis, and I'll endeavor to incorporate them into each of the statistical approaches we discuss.

In the previous chapter we discussed a range of methods for inputting or importing data into R. Unfortunately, in the real world your data is rarely usable in the format in which you first get it. In the next chapter we look at ways to transform and massage our data into a state that's more useful and conducive to analysis.

6 Basic graphs

This chapter covers

- Bar, box, and dot plots
- Pie and fan charts
- Histograms and kernel density plots

Whenever we analyze data, the first thing that we should do is *look* at it. For each variable, what are the most common values? How much variability is present? Are there any unusual observations? R provides a wealth of functions for visualizing data. In this chapter, we'll look at graphs that help you understand a single categorical or continuous variable. This topic includes

- Visualizing the distribution of variable
- Comparing groups on an outcome variable

In both cases, the variable could be continuous (for example, car mileage as miles per gallon) or categorical (for example, treatment outcome as none, some, or marked). In later chapters, we'll explore graphs that display bivariate and multivariate relationships among variables.

In the following sections, we'll explore the use of bar plots, pie charts, fan charts, histograms, kernel density plots, box plots, violin plots, and dot plots. Some of these may be familiar to you, whereas others (such as fan plots or violin plots) may be new to you. Our goal, as always, is to understand your data better and to communicate this understanding to others.

Let's start with bar plots.

6.1 Bar plots

Bar plots display the distribution (frequencies) of a categorical variable through vertical or horizontal bars. In its simplest form, the format of the `barplot()` function is

```
barplot(height)
```

where *`height`* is a vector or matrix.

In the following examples, we'll plot the outcome of a study investigating a new treatment for rheumatoid arthritis. The data are contained in the `Arthritis` data frame distributed with the `vcd` package. Because the `vcd` package isn't included in the default R installation, be sure to download and install it before first use (`install.packages("vcd")`).

Note that the `vcd` package isn't needed to create bar plots. We're loading it in order to gain access to the `Arthritis` dataset. But we'll need the `vcd` package when creating spinogram, which are described in section 6.1.5.

6.1.1 Simple bar plots

If *`height`* is a vector, the values determine the heights of the bars in the plot and a vertical bar plot is produced. Including the option `horiz=TRUE` produces a horizontal bar chart instead. You can also add annotating options. The `main` option adds a plot title, whereas the `xlab` and `ylab` options add x-axis and y-axis labels, respectively.

In the Arthritis study, the variable `Improved` records the patient outcomes for individuals receiving a placebo or drug.

```
> library(vcd)
> counts <- table(Arthritis$Improved)
> counts
  None    Some  Marked
    42      14      28
```

Here, we see that 28 patients showed marked improvement, 14 showed some improvement, and 42 showed no improvement. We'll discuss the use of the `table()` function to obtain cell counts more fully in chapter 7.

You can graph the variable `counts` using a vertical or horizontal bar plot. The code is provided in the following listing and the resulting graphs are displayed in figure 6.1.

Listing 6.1 Simple bar plots

```
barplot(counts,                                      <- Simple bar plot
        main="Simple Bar Plot",
        xlab="Improvement", ylab="Frequency")

barplot(counts,                                      <- Horizontal bar plot
        main="Horizontal Bar Plot",
        xlab="Frequency", ylab="Improvement",
        horiz=TRUE)
```

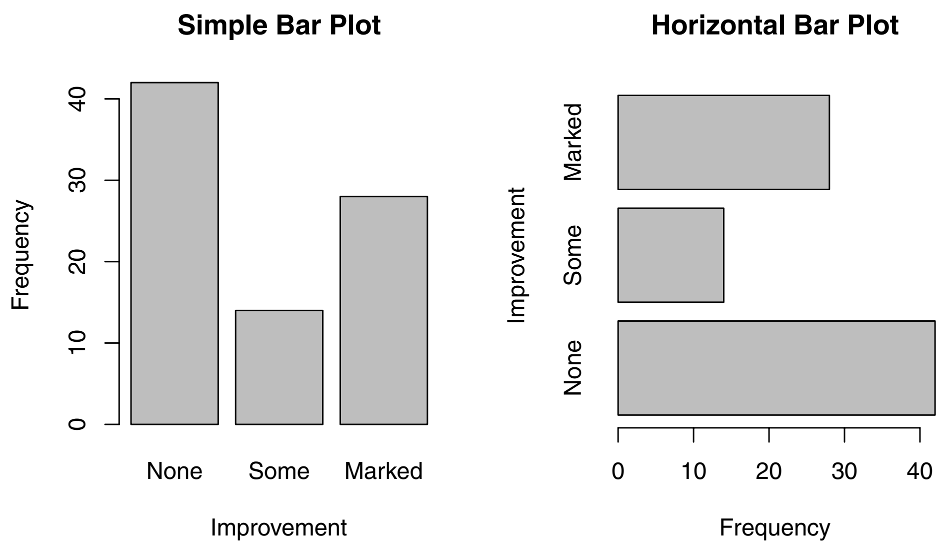


Figure 6.1 **Simple vertical and horizontal bar charts**

TIP If the categorical variable to be plotted is a factor or ordered factor, you can create a vertical bar plot quickly with the `plot()` function. Because `Arthritis$Improved` is a factor, the code

```
plot(Arthritis$Improved, main="Simple Bar Plot",
     xlab="Improved", ylab="Frequency")
plot(Arthritis$Improved, horiz=TRUE, main="Horizontal Bar Plot",
     xlab="Frequency", ylab="Improved")
```

will generate the same bar plots as those in listing 6.1, but without the need to tabulate values with the `table()` function.

What happens if you have long labels? In section 6.1.4, you'll see how to tweak labels so that they don't overlap.

6.1.2 *Stacked and grouped bar plots*

If height is a matrix rather than a vector, the resulting graph will be a stacked or grouped bar plot. If `beside=FALSE` (the default), then each column of the matrix produces a bar in the plot, with the values in the column giving the heights of stacked "sub-bars." If `beside=TRUE`, each column of the matrix represents a group, and the values in each column are juxtaposed rather than stacked.

Consider the cross-tabulation of treatment type and improvement status:

```
> library(vcd)
> counts <- table(Arthritis$Improved, Arthritis$Treatment)
> counts
        Treatment
```

```
Improved Placebo Treated
  None          29      13
  Some           7       7
  Marked         7      21
```

You can graph the results as either a stacked or a grouped bar plot (see the next listing). The resulting graphs are displayed in figure 6.2.

Listing 6.2 Stacked and grouped bar plotsw

```
barplot(counts,                                      Stacked bar plot
        main="Stacked Bar Plot",
        xlab="Treatment", ylab="Frequency",
        col=c("red", "yellow","green"),
        legend=rownames(counts))

barplot(counts,                                      Grouped bar plot
        main="Grouped Bar Plot",
        xlab="Treatment", ylab="Frequency",
        col=c("red", "yellow", "green"),
        legend=rownames(counts), beside=TRUE)
```

The first barplot function produces a stacked bar plot, whereas the second produces a grouped bar plot. We've also added the col option to add color to the bars plotted. The legend.text parameter provides bar labels for the legend (which are only useful when height is a matrix).

In chapter 3, we covered ways to format and place the legend to maximum benefit. See if you can rearrange the legend to avoid overlap with the bars.

6.1.3 Mean bar plots

Bar plots needn't be based on counts or frequencies. You can create bar plots that represent means, medians, standard deviations, and so forth by using the aggregate

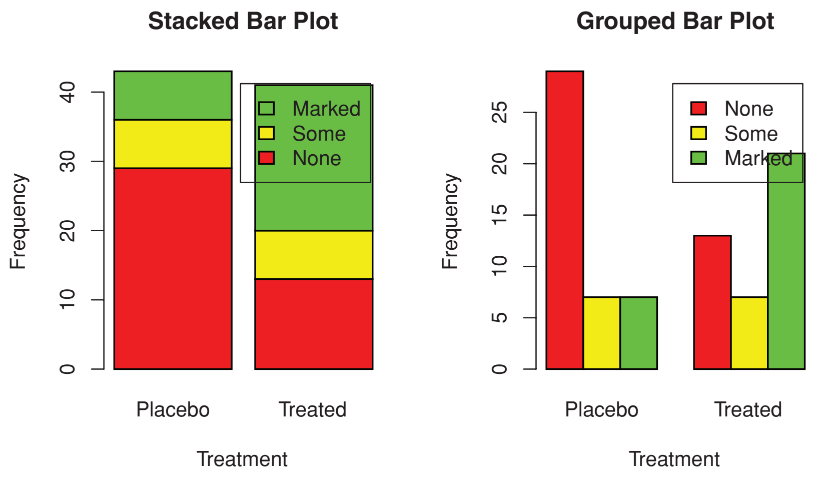


Figure 6.2 Stacked and grouped bar plots

function and passing the results to the barplot() function. The following listing shows an example, which is displayed in figure 6.3.

Listing 6.3 Bar plot for sorted mean values

```
> states <- data.frame(state.region, state.x77)
> means <- aggregate(states$Illiteracy, by=list(state.region), FUN=mean)
> means
        Group.1    x
1     Northeast 1.00
2         South 1.74
3 North Central 0.70
4          West 1.02
> means <- means[order(means$x),]                    ← (1) Means sorted smallest to largest
> means
        Group.1    x
3 North Central 0.70
1     Northeast 1.00
4          West 1.02
2         South 1.74
> barplot(means$x, names.arg=means$Group.1)
> title("Mean Illiteracy Rate")                      ← (2) Title added
```

Listing 6.3 sorts the means from smallest to largest (1). Also note that use of the `title()` function (2) is equivalent to adding the main option in the plot call. `means$x` is the vector containing the heights of the bars, and the option `names.arg=means$Group.1` is added to provide labels.

You can take this example further. The bars can be connected with straight line segments using the `lines()` function. You can also create mean bar plots with superimposed confidence intervals using the `barplot2()` function in the `gplots` package. See "barplot2: Enhanced Bar Plots" on the R Graph Gallery website (http://addictedtor.free.fr/graphiques) for an example.

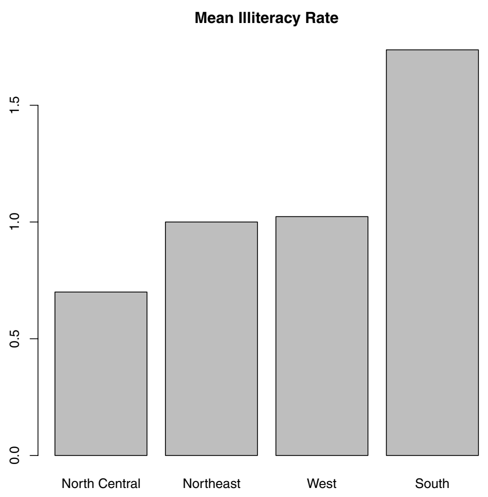


Figure 6.3 Bar plot of mean illiteracy rates for US regions sorted by rate

6.1.4 Tweaking bar plots

There are several ways to tweak the appearance of a bar plot. For example, with many bars, bar labels may start to overlap. You can decrease the font size using the `cex.names` option. Specifying values smaller than 1 will shrink the size of the labels. Optionally, the `names.arg` argument allows you to specify a character vector of names used to label the bars. You can also use graphical parameters to help text spacing. An example is given in the following listing with the output displayed in figure 6.4.

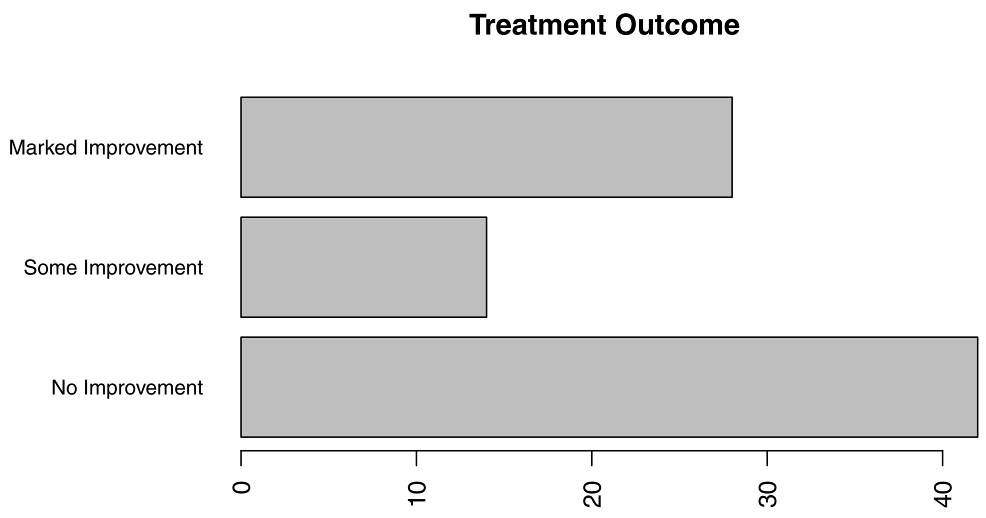


Figure 6.4 Horizontal bar plot with tweaked labels

Listing 6.4 Fitting labels in a bar plot

```
par(mar=c(5,8,4,2))
par(las=2)
counts <- table(Arthritis$Improved)

barplot(counts,
           main="Treatment Outcome",
           horiz=TRUE, cex.names=0.8,
        names.arg=c("No Improvement", "Some Improvement",
                    "Marked Improvement"))
```

In this example, we've rotated the bar labels (with `las=2`), changed the label text, and both increased the size of the y margin (with `mar`) and decreased the font size in order to fit the labels comfortably (using `cex.names=0.8`). The `par()` function allows you to make extensive modifications to the graphs that R produces by default. See chapter 3 for more details.

6.1.5 *Spinograms*

Before finishing our discussion of bar plots, let's take a look at a specialized version called a *spinogram*. In a spinogram, a stacked bar plot is rescaled so that the height of each bar is 1 and the segment heights represent proportions. Spinograms are created through the `spine()` function of the `vcd` package. The following code produces a simple spinogram:

```
library(vcd)
attach(Arthritis)
counts <- table(Treatment, Improved)
spine(counts, main="Spinogram Example")
detach(Arthritis)
```

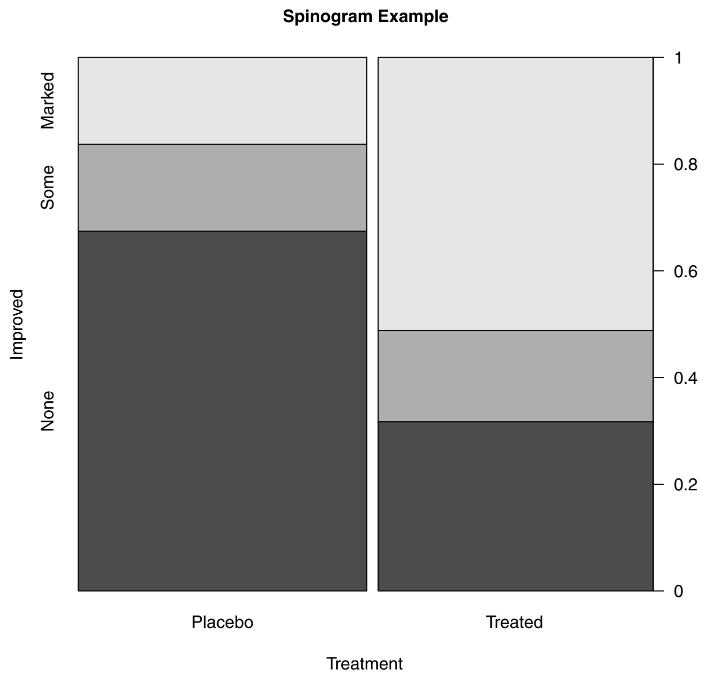


Figure 6.5 Spinogram of arthritis treatment outcome

The output is provided in figure 6.5. The larger percentage of patients with marked improvement in the Treated condition is quite evident when compared with the Placebo condition.

In addition to bar plots, pie charts are a popular vehicle for displaying the distribution of a categorical variable. We consider them next.

6.2 Pie charts

Whereas pie charts are ubiquitous in the business world, they're denigrated by most statisticians, including the authors of the R documentation. They recommend bar or dot plots over pie charts because people are able to judge length more accurately than volume. Perhaps for this reason, the pie chart options in R are quite limited when compared with other statistical software.

Pie charts are created with the function

```
pie(x, labels)
```

where *x* is a non-negative numeric vector indicating the area of each slice and `labels` provides a character vector of slice labels. Four examples are given in the next listing; the resulting plots are provided in figure 6.6.

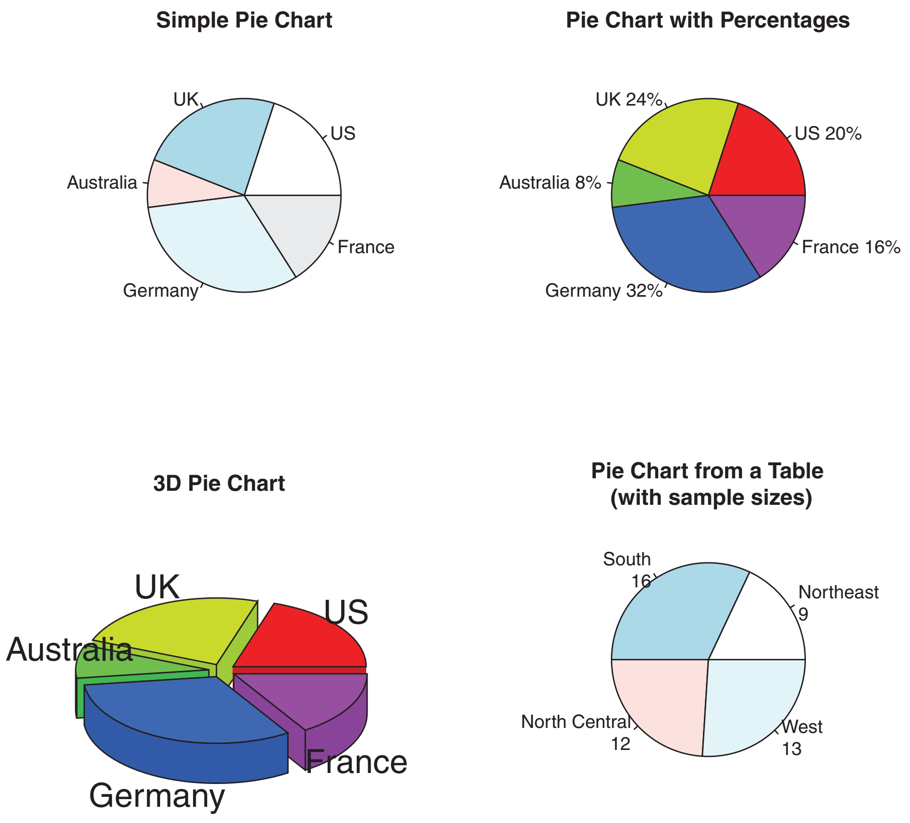


Figure 6.6 Pie chart examples

Listing 6.5 Pie charts

```
par(mfrow=c(2, 2))
slices <- c(10, 12,4, 16, 8)
lbls <- c("US", "UK", "Australia", "Germany", "France")

pie(slices, labels = lbls,
    main="Simple Pie Chart")

pct <- round(slices/sum(slices)*100)
lbls2 <- paste(lbls, " ", pct, "%", sep="")
pie(slices, labels=lbls2, col=rainbow(length(lbls2)),
    main="Pie Chart with Percentages")

library(plotrix)
pie3D(slices, labels=lbls,explode=0.1,
      main="3D Pie Chart ")

mytable <- table(state.region)
lbls3 <- paste(names(mytable), "\n", mytable, sep="")
pie(mytable, labels = lbls3,
    main="Pie Chart from a Table\n (with sample sizes)")
```

1 Combine four graphs into one

2 Add percentages to pie chart

3 Create chart from table

First you set up the plot so that four graphs are combined into one ❶. (Combining multiple graphs is covered in chapter 3.) Then you input the data that will be used for the first three graphs.

For the second pie chart ❷, you convert the sample sizes to percentages and add the information to the slice labels. The second pie chart also defines the colors of the slices using the `rainbow()` function, described in chapter 3. Here `rainbow(length(lbls2))` resolves to `rainbow(5)`, providing five colors for the graph.

The third pie chart is a 3D chart created using the `pie3D()` function from the `plotrix` package. Be sure to download and install this package before using it for the first time. If statisticians dislike pie charts, they positively despise 3D pie charts (although they may secretly find them pretty). This is because the 3D effect adds no additional insight into the data and is considered distracting eye candy.

The fourth pie chart demonstrates how to create a chart from a table ❸. In this case, you count the number of states by US region, and append the information to the labels before producing the plot.

Pie charts make it difficult to compare the values of the slices (unless the values are appended to the labels). For example, looking at the simple pie chart, can you tell how the US compares to Germany? (If you can, you're more perceptive than I am.) In an attempt to improve on this situation, a variation of the pie chart, called a fan plot, has been developed. The fan plot (Lemon & Tyagi, 2009) provides the user with a way to display both relative quantities and differences. In R, it's implemented through the `fan.plot()` function in the `plotrix` package.

Consider the following code and the resulting graph (figure 6.7):

```
library(plotrix)
slices <- c(10, 12,4, 16, 8)
lbls <- c("US", "UK", "Australia", "Germany", "France")
fan.plot(slices, labels = lbls, main="Fan Plot")
```

In a fan plot, the slices are rearranged to overlap each other and the radii have been modified so that each slice is visible. Here you can see that Germany is the largest slice

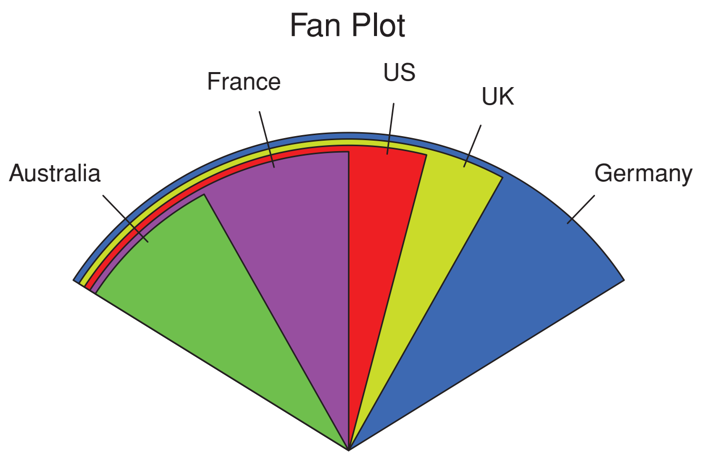


Figure 6.7 A fan plot of the country data

and that the US slice is roughly 60 percent as large. France appears to be half as large as Germany and twice as large as Australia. Remember that the *width* of the slice and not the radius is what's important here.

As you can see, it's much easier to determine the relative sizes of the slice in a fan plot than in a pie chart. Fan plots haven't caught on yet, but they're new. Now that we've covered pie and fan charts, let's move on to histograms. Unlike bar plots and pie charts, histograms describe the distribution of a continuous variable.

6.3 Histograms

Histograms display the distribution of a continuous variable by dividing up the range of scores into a specified number of bins on the x-axis and displaying the frequency of scores in each bin on the y-axis. You can create histograms with the function

```
hist(x)
```

where *x* is a numeric vector of values. The option `freq=FALSE` creates a plot based on probability densities rather than frequencies. The `breaks` option controls the number of bins. The default produces equally spaced breaks when defining the cells of the histogram. Listing 6.6 provides the code for four variations of a histogram; the results are plotted in figure 6.8.

Listing 6.6 Histograms

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

hist(mtcars$mpg)                            <- 1 Simple histogram

hist(mtcars$mpg,                                       <- 2 With specified
     breaks=12,                                             bins and color
     col="red",
     xlab="Miles Per Gallon",
     main="Colored histogram with 12 bins")

hist(mtcars$mpg,                                      <- 3 With rug plot
     freq=FALSE,
     breaks=12,
     col="red",
     xlab="Miles Per Gallon",
     main="Histogram, rug plot, density curve")
rug(jitter(mtcars$mpg))
lines(density(mtcars$mpg), col="blue", lwd=2)

x <- mtcars$mpg                               <- 4 With normal curve
h<-hist(x,                                         and frame
        breaks=12,
        col="red",
        xlab="Miles Per Gallon",
        main="Histogram with normal curve and box")
xfit<-seq(min(x), max(x), length=40)
yfit<-dnorm(xfit, mean=mean(x), sd=sd(x))
yfit <- yfit*diff(h$mids[1:2])*length(x)
lines(xfit, yfit, col="blue", lwd=2)
box()
```

The first histogram ❶ demonstrates the default plot when no options are specified. In this case, five bins are created, and the default axis labels and titles are printed. For the second histogram ❷, you've specified 12 bins, a red fill for the bars, and more attractive and informative labels and title.

The third histogram ❸ maintains the colors, bins, labels, and titles as the previous plot, but adds a density curve and rug plot overlay. The density curve is a kernel density estimate and is described in the next section. It provides a smoother description of the distribution of scores. You use the `lines()` function to overlay this curve in a blue color and a width that's twice the default thickness for lines. Finally, a rug plot is a one-dimensional representation of the actual data values. If there are many tied values, you can jitter the data on the rug plot using code like the following:

```
rug(jitter(mtcars$mpag, amount=0.01))
```

This will add a small random value to each data point (a uniform random variate between ±`amount`), in order to avoid overlapping points.

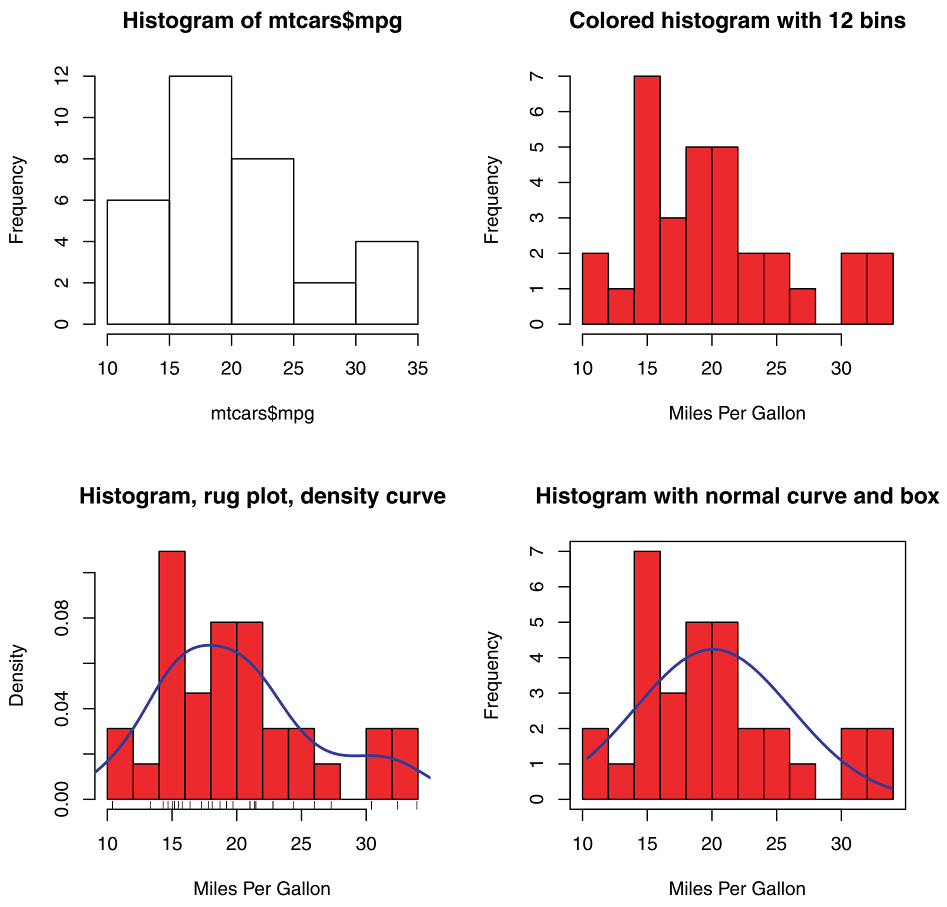


Figure 6.8 Histograms examples

The fourth histogram (4) is similar to the second but has a superimposed normal curve and a box around the figure. The code for superimposing the normal curve comes from a suggestion posted to the R-help mailing list by Peter Dalgaard. The surrounding box is produced by the `box()` function.

6.4 Kernel density plots

In the previous section, you saw a kernel density plot superimposed on a histogram. Technically, kernel density estimation is a nonparametric method for estimating the probability density function of a random variable. Although the mathematics are beyond the scope of this text, in general kernel density plots can be an effective way to view the distribution of a continuous variable. The format for a density plot (that's not being superimposed on another graph) is

```
plot(density(x))
```

where *x* is a numeric vector. Because the `plot()` function begins a new graph, use the `lines()` function (listing 6.6) when superimposing a density curve on an existing graph.

Two kernel density examples are given in the next listing, and the results are plotted in figure 6.9.

Listing 6.7 Kernel density plots

```
par(mfrow=c(2,1))
d <- density(mtcars$mpg)

plot(d)

d <- density(mtcars$mpg)
plot(d, main="Kernel Density of Miles Per Gallon")
polygon(d, col="red", border="blue")
rug(mtcars$mpg, col="brown")
```

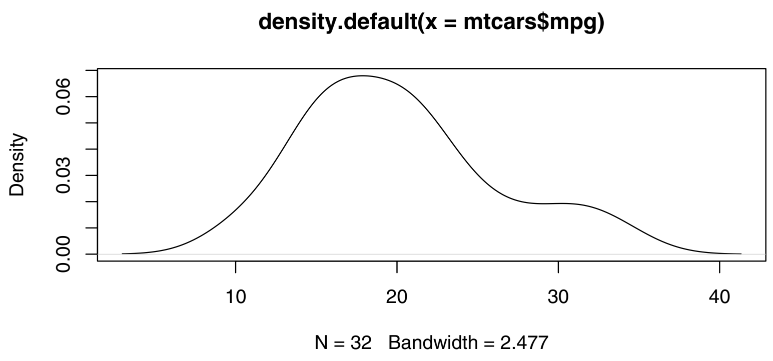
In the first plot, you see the minimal graph created with all the defaults in place. In the second plot, you add a title, color the curve blue, fill the area under the curve with solid red, and add a brown rug. The `polygon()` function draws a polygon whose vertices are given by x and y (provided by the `density()` function in this case).

Kernel density plots can be used to compare groups. This is a highly underutilized approach, probably due to a general lack of easily accessible software. Fortunately, the `sm` package fills this gap nicely.

The `sm.density.compare()` function in the `sm` package allows you to superimpose the kernel density plots of two or more groups. The format is

```
sm.density.compare(x, factor)
```

where x is a numeric vector and `factor` is a grouping variable. Be sure to install the `sm` package before first use. An example comparing the mpg of cars with 4, 6, or 8 cylinders is provided in listing 6.8.



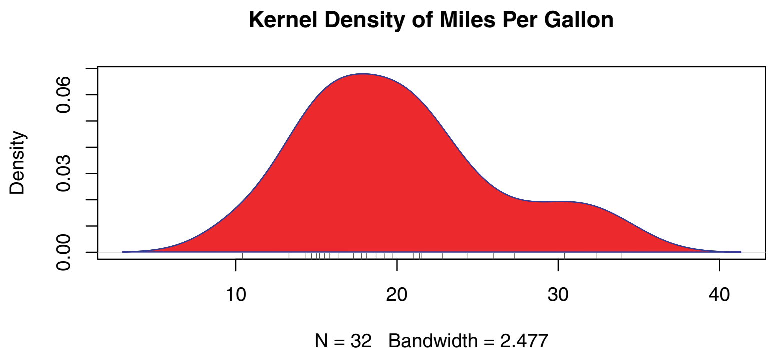


Figure 6.9 Kernel density plots

Listing 6.8 Comparative kernel density plots

```
par(lwd=2)
library(sm)
attach(mtcars)

cyl.f <- factor(cyl, levels= c(4,6,8),
              labels = c("4 cylinder", "6 cylinder",
                         "8 cylinder"))

sm.density.compare(mpg, cyl, xlab="Miles Per Gallon")
title(main="MPG Distribution by Car Cylinders")

colfill<-c(2:(1+length(levels(cyl.f))))
legend(locator(1), levels(cyl.f), fill=colfill)

detach(mtcars)
```

1 Double width of plotted lines

2 Create grouping factor

3 Plot densities

4 Add legend via mouse click

The `par()` function is used to double the width of the plotted lines `(lwd=2)` so that they'd be more readable in this book 1. The `sm` packages is loaded and the `mtcars` data frame is attached.

In the mtcars data frame ❷, the variable cyl is a numeric variable coded 4, 6, or 8. cyl is transformed into a factor, named cyl.f, in order to provide value labels for the plot. The sm.density.compare() function creates the plot ❸ and a title() statement adds a main title.

Finally, you add a legend to improve interpretability ❹. (Legends are covered in chapter 3.) First, a vector of colors is created. Here colfill is c(2,3,4). Then a legend is added to the plot via the legend() function. The locator(1) option indicates that you'll place the legend interactively by clicking on the graph where you want the legend to appear. The second option provides a character vector of the labels. The third option assigns a color from the vector colfill to each level of cyl.f. The results are displayed in figure 6.10.

As you can see, overlapping kernel density plots can be a powerful way to compare groups of observations on an outcome variable. Here you can see both the shapes of the distribution of scores for each group and the amount of overlap between groups. (The moral of the story is that my next car will have four cylinders—or a battery.)

Box plots are also a wonderful (and more commonly used) graphical approach to visualizing distributions and differences among groups. We'll discuss them next.

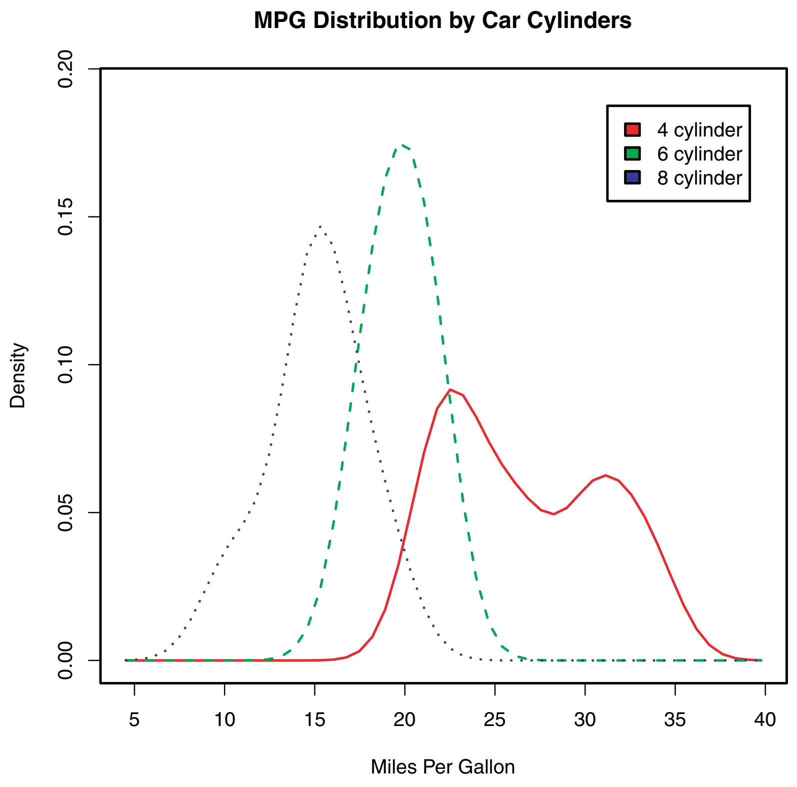


Figure 6.10 **Kernel density plots of mpg by number of cylinders**

6.5 Box plots

A “box-and-whiskers” plot describes the distribution of a continuous variable by plotting its five-number summary: the minimum, lower quartile (25th percentile), median (50th percentile), upper quartile (75th percentile), and maximum. It can also display observations that may be outliers (values outside the range of ± 1.5*IQR, where IQR is the interquartile range defined as the upper quartile minus the lower quartile). For example:

```
boxplot(mtcars$mpg, main="Box plot", ylab="Miles per Gallon")
```

produces the plot shown in figure 6.11. I added annotations by hand to illustrate the components.

By default, each whisker extends to the most extreme data point, which is no more than the 1.5 times the interquartile range for the box. Values outside this range are depicted as dots (not shown here).

For example, in our sample of cars the median mpg is 19.2, 50 percent of the scores fall between 15.3 and 22.8, the smallest value is 10.4, and the largest value is 33.9. How did I read this so precisely from the graph? Issuing `boxplot.stats(mtcars$mpg)`

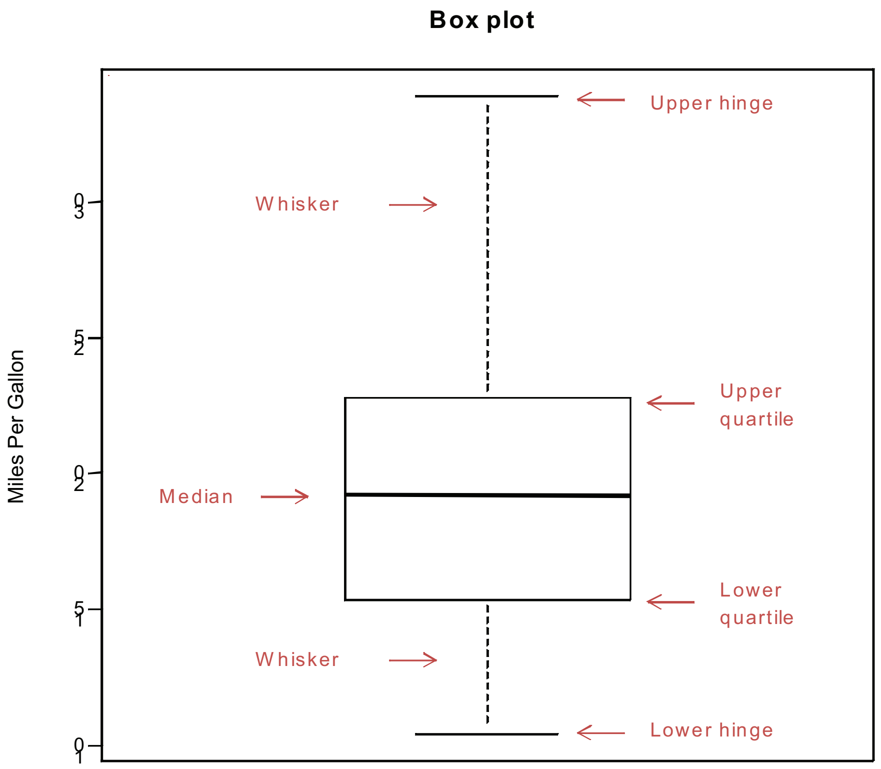


Figure 6.11 Box plot with annotations added by hand

prints the statistics used to build the graph (in other words, I cheated). There doesn't appear to be any outliers, and there is a mild positive skew (the upper whisker is longer than the lower whisker).

6.5.1 Using parallel box plots to compare groups

Box plots can be created for individual variables or for variables by group. The format is

```
boxplot(formula, data=dataframe)
```

where *formula* is a formula and *dataframe* denotes the data frame (or list) providing the data. An example of a formula is `y ~ A`, where a separate box plot for numeric variable `y` is generated for each value of categorical variable `A`. The formula `y ~ A*B` would produce a box plot of numeric variable `y`, for each combination of levels in categorical variables `A` and `B`.

Adding the option `varwidth=TRUE` will make the box plot widths proportional to the square root of their sample sizes. Add `horizontal=TRUE` to reverse the axis orientation.

In the following code, we revisit the impact of four, six, and eight cylinders on auto mpg with parallel box plots. The plot is provided in figure 6.12.

```
boxplot(mpg ~ cyl, data=mtcars,
        main="Car Mileage Data",
        xlab="Number of Cylinders",
        ylab="Miles Per Gallon")
```

You can see in figure 6.12 that there's a good separation of groups based on gas mileage. You can also see that the distribution of mpg for six-cylinder cars is more symmetrical than for the other two car types. Cars with four cylinders show the greatest spread (and positive skew) of mpg scores, when compared with six- and eight-cylinder cars. There's also an outlier in the eight-cylinder group.

Box plots are very versatile. By adding `notch=TRUE`, you get *notched* box plots. If two boxes' notches don't overlap, there's strong evidence that their medians differ (Chambers et al., 1983, p. 62). The following code will create notched box plots for our mpg example:

```
boxplot(mpg ~ cyl, data=mtcars,
        notch=TRUE,
        varwidth=TRUE,
        col="red",
        main="Car Mileage Data",
        xlab="Number of Cylinders",
        ylab="Miles Per Gallon")
```

The `col` option fills the box plots with a red color, and `varwidth=TRUE` produces box plots with widths that are proportional to their sample sizes.

You can see in figure 6.13 that the median car mileage for four-, six-, and eight-cylinder cars differ. Mileage clearly decreases with number of cylinders.

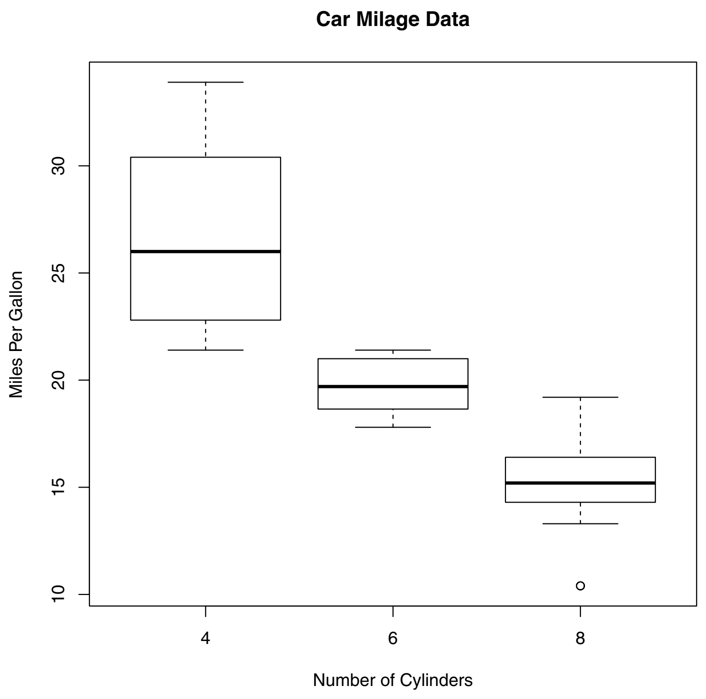


Figure 6.12 Box plots of car mileage versus number of cylinders

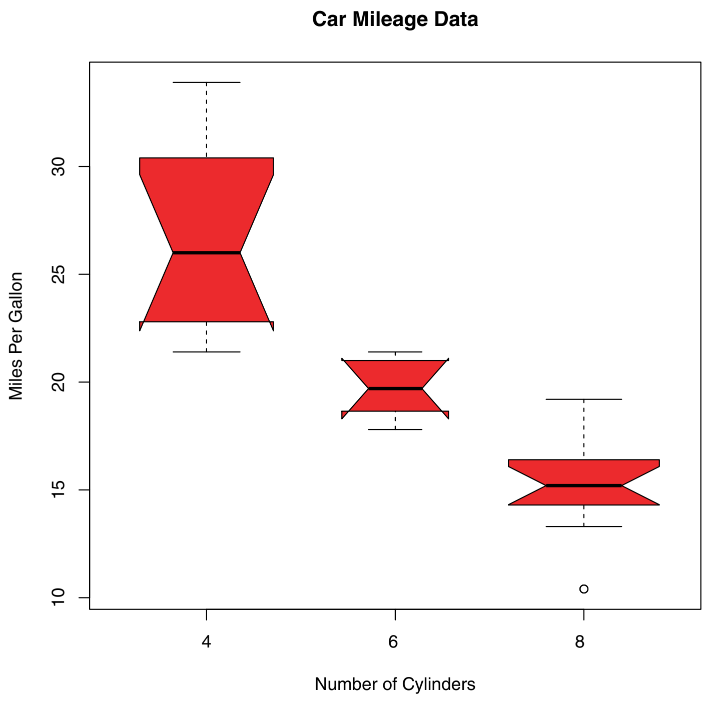


Figure 6.13 Notched box plots for car mileage versus number of cylinders

Finally, you can produce box plots for more than one grouping factor. Listing 6.9 provides box plots for mpg versus the number of cylinders and transmission type in an automobile. Again, you use the `col` option to fill the box plots with color. Note that colors recycle. In this case, there are six box plots and only two specified colors, so the colors repeat three times.

Listing 6.9 Box plots for two crossed factors

```
mtcars$cyl.f <- factor(mtcars$cyl,
                       levels=c(4,6,8),
                       labels=c("4","6","8"))
```
Create factor for # of cylinders

```
mtcars$am.f <- factor(mtcars$am,
                      levels=c(0,1),
                      labels=c("auto", "standard"))
```
Create factor for transmission type

```
boxplot(mpg ~ am.f *cyl.f,
        data=mtcars,
        varwidth=TRUE,
        col=c("gold","darkgreen"),
        main="MPG Distribution by Auto Type",
        xlab="Auto Type")
```
Generate box plot

The plot is provided in figure 6.14.

From figure 6.14 it's again clear that median mileage decreases with cylinder number. For four- and six-cylinder cars, mileage is higher for standard transmissions. But for eight-cylinder cars there doesn't appear to be a difference. You can also see

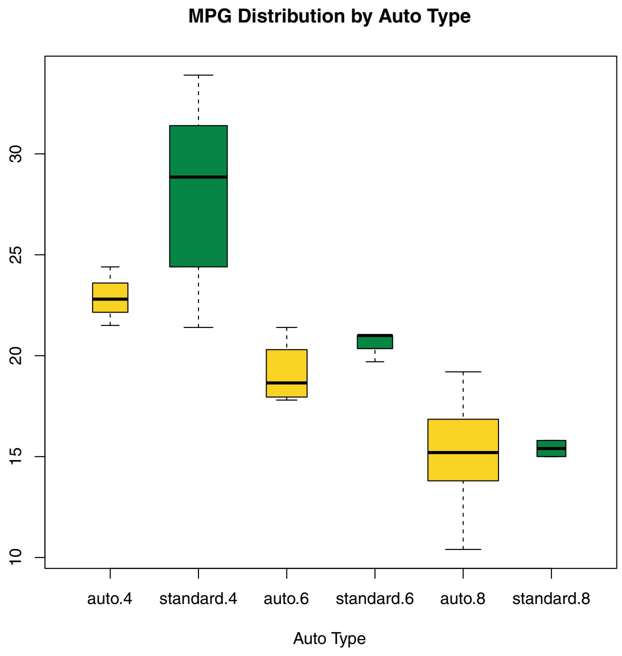


Figure 6.14 Box plots for car mileage versus transmission type and number of cylinders

from the widths of the box plots that standard four-cylinder and automatic eight-cylinder cars are the most common in this dataset.

6.5.2 Violin plots

Before we end our discussion of box plots, it's worth examining a variation called a violin plot. A violin plot is a combination of a box plot and a kernel density plot. You can create one using the `vioplot()` function from the `vioplot` package. Be sure to install the `vioplot` package before first use.

The format for the `vioplot()` function is

```
vioplot(x1, x2, … , names=, col=)
```

where *x1*, *x2*, … represent one or more numeric vectors to be plotted (one violin plot will be produced for each vector). The `names` parameter provides a character vector of labels for the violin plots, and `col` is a vector specifying the colors for each violin plot. An example is given in the following listing.

Listing 6.10 Violin plots

```
library(vioplot)
x1 <- mtcars$mpg[mtcars$cyl==4]
x2 <- mtcars$mpg[mtcars$cyl==6]
x3 <- mtcars$mpg[mtcars$cyl==8]
vioplot(x1, x2, x3,
        names=c("4 cyl", "6 cyl", "8 cyl"),
        col="gold")
title("Violin Plots of Miles Per Gallon")
```

Note that the `vioplot()` function requires you to separate the groups to be plotted into separate variables. The results are displayed in figure 6.15.

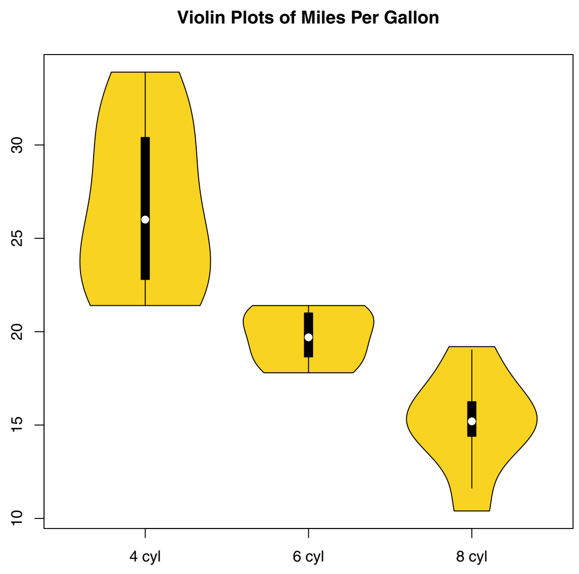


Figure 6.15 Violin plots of mpg versus number of cylinders

Violin plots are basically kernel density plots superimposed in a mirror image fashion over box plots. Here, the white dot is the median, the black boxes range from the lower to the upper quartile, and the thin black lines represent the whiskers. The outer shape provides the kernel density plots. Violin plots haven't really caught on yet. Again, this may be due to a lack of easily accessible software. Time will tell.

We'll end this chapter with a look at dot plots. Unlike the graphs you've seen previously, dot plots plot every value for a variable.

6.6 Dot plots

Dot plots provide a method of plotting a large number of labeled values on a simple horizontal scale. You create them with the `dotchart()` function, using the format

```
dotchart(x, labels=)
```

where *x* is a numeric vector and `labels` specifies a vector that labels each point. You can add a `groups` option to designate a factor specifying how the elements of *x* are grouped. If so, the option `gcolor` controls the color of the groups label and `cex` controls the size of the labels. Here's an example with the `mtcars` dataset:

```
dotchart(mtcars$mpg, labels=row.names(mtcars), cex=.7,
         main="Gas Mileage for Car Models",
         xlab="Miles Per Gallon")
```

The resulting plot is given in figure 6.16.

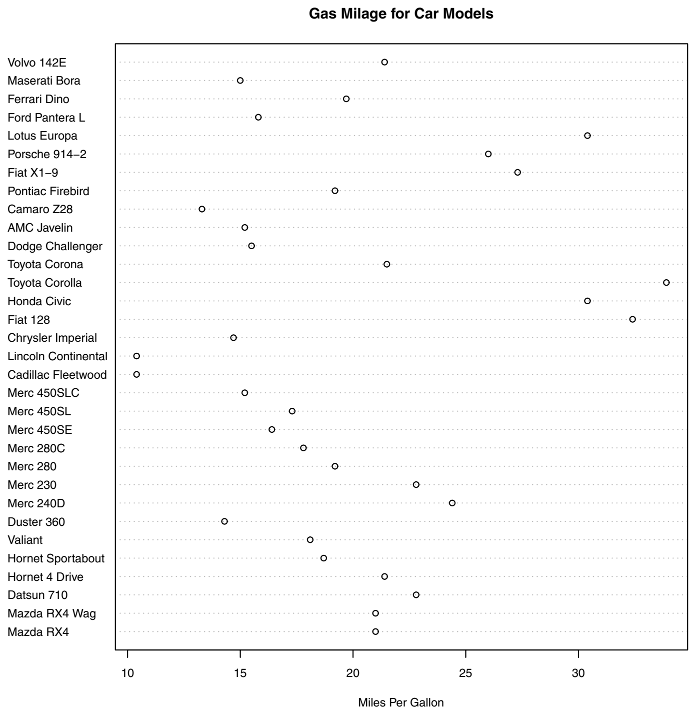


Figure 6.16 Dot plot of mpg for each car model

The graph in figure 6.16 allows you to see the mpg for each make of car on the same horizontal axis. Dot plots typically become most interesting when they're sorted and grouping factors are distinguished by symbol and color. An example is given in the following listing.

Listing 6.11 Dot plot grouped, sorted, and colored

```
x <- mtcars[order(mtcars$mpg),]
x$cyl <- factor(x$cyl)
x$color[x$cyl==4] <- "red"
x$color[x$cyl==6] <- "blue"
x$color[x$cyl==8] <- "darkgreen"
dotchart(x$mpg,
         labels = row.names(x),
         cex=.7,
         groups = x$cyl,
         gcolor = "black",
         color = x$color,
         pch=19,
         main = "Gas Mileage for Car Models\ngrouped by cylinder",
         xlab = "Miles Per Gallon")
```

In this example, the data frame `mtcars` is sorted by mpg (lowest to highest) and saved as data frame `x`. The numeric vector `cyl` is transformed into a factor. A character vector (`color`) is added to data frame `x` and contains the values `"red"`, `"blue"`, or `"darkgreen"` depending on the value of `cyl`. In addition, the labels for the data points are taken from the row names of the data frame (car makes). Data points are grouped by number of cylinders. The numbers 4, 6, and 8 are printed in black. The color of the points and labels are derived from the `color` vector, and points are represented by filled circles. The code produces the graph in figure 6.17.

In figure 6.17, a number of features become evident for the first time. Again, you see an increase in gas mileage as the number of cylinders decrease. But you also see exceptions. For example, the Pontiac Firebird, with eight cylinders, gets higher gas mileage than the Mercury 280C and the Valiant, each with six cylinders. The Hornet 4 Drive, with six cylinders, gets the same miles per gallon as the Volvo 142E, which has four cylinders. It's also clear that the Toyota Corolla gets the best gas mileage by far, whereas the Lincoln Continental and Cadillac Fleetwood are outliers on the low end.

You can gain significant insight from a dot plot in this example because each point is labeled, the value of each point is inherently meaningful, and the points are arranged in a manner that promotes comparisons. But as the number of data points increase, the utility of the dot plot decreases.

NOTE There are many variations of the dot plot. Jacoby (2006) provides a very informative discussion of the dot plot and provides R code for innovative applications. Additionally, the `Hmisc` package offers a dot plot function (aptly named `dotchart2`) with a number of additional features.

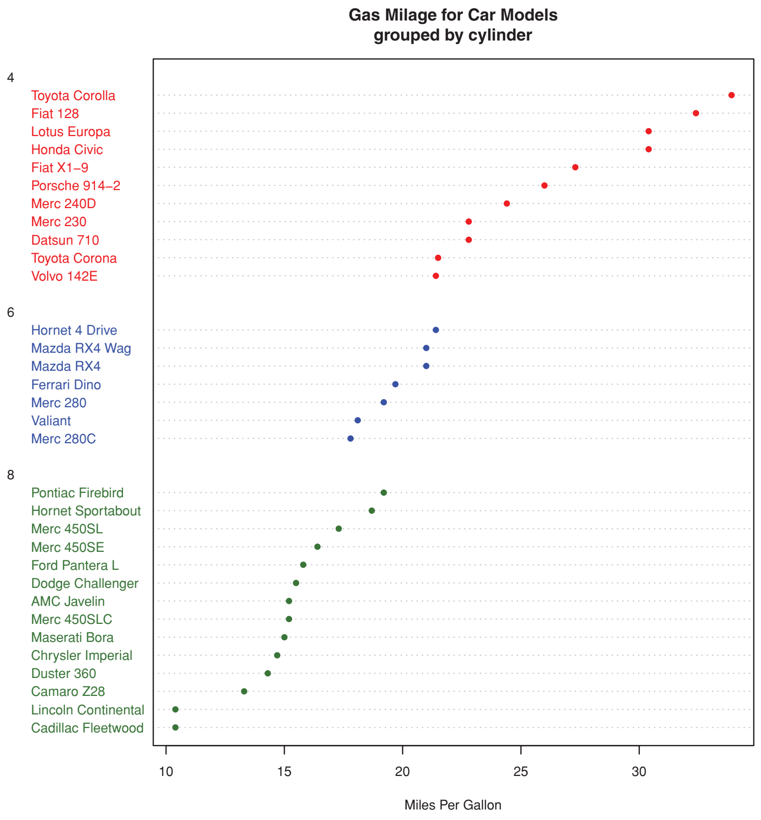


Figure 6.17 Dot plot of mpg for car models grouped by number of cylinders

6.7 Summary

In this chapter, we learned how to describe continuous and categorical variables. We saw how bar plots and (to a lesser extent) pie charts can be used to gain insight into the distribution of a categorical variable, and how stacked and grouped bar charts can help us understand how groups differ on a categorical outcome. We also explored how histograms, kernel density plots, box plots, rug plots, and dot plots can help us visualize the distribution of continuous variables. Finally, we explored how overlapping kernel density plots, parallel box plots, and grouped dot plots can help you visualize group differences on a continuous outcome variable.

In later chapters, we'll extend this univariate focus to include bivariate and multivariate graphical methods. You'll see how to visually depict relationships among many variables at once, using such methods as scatter plots, multigroup line plots, mosaic plots, correlograms, lattice graphs, and more.

In the next chapter, we'll look at basic statistical methods for describing distributions and bivariate relationships numerically, as well as inferential methods for evaluating whether relationships among variables exist or are due to sampling error.

11 Intermediate graphs

This chapter covers

- Visualizing bivariate and multivariate relationships
- Working with scatter and line plots
- Understanding correlograms
- Using mosaic and association plots

In chapter 6 (basic graphs), we considered a wide range of graph types for displaying the distribution of single categorical or continuous variables. Chapter 8 (regression) reviewed graphical methods that are useful when predicting a continuous outcome variable from a set of predictor variables. In chapter 9 (analysis of variance), we considered techniques that are particularly useful for visualizing how groups differ on a continuous outcome variable. In many ways, the current chapter is a continuation and extension of the topics covered so far.

In this chapter, we'll focus on graphical methods for displaying relationships between two variables (bivariate relationships) and between many variables (multivariate relationships). For example:

- What's the relationship between automobile mileage and car weight? Does it vary by the number of cylinders the car has?
- How can you picture the relationships among an automobile's mileage, weight, displacement, and rear axle ratio in a single graph?

- When plotting the relationship between two variables drawn from a large dataset (say 10,000 observations), how can you deal with the massive overlap of data points you're likely to see? In other words, what do you do when your graph is one big smudge?
- How can you visualize the multivariate relationships among three variables at once (given a 2D computer screen or sheet of paper, and a budget slightly less than that for *Avatar*)?
- How can you display the growth of several trees over time?
- How can you visualize the correlations among a dozen variables in a single graph? How does it help you to understand the structure of your data?
- How can you visualize the relationship of class, gender, and age with passenger survival on the *Titanic*? What can you learn from such a graph?

These are the types of questions that can be answered with the methods described in this chapter. The datasets that we'll use are examples of what's possible. It's the general techniques that are most important. If the topic of automobile characteristics or tree growth isn't interesting to you, plug in your own data!

We'll start with scatter plots and scatter plot matrices. Then, we'll explore line charts of various types. These approaches are well known and widely used in research. Next, we'll review the use of correlograms for visualizing correlations and mosaic plots for visualizing multivariate relationships among categorical variables. These approaches are also useful but much less well known among researchers and data analysts. You'll see examples of how you can use each of these approaches to gain a better understanding of your data and communicate these findings to others.

11.1 Scatter plots

As you've seen in previous chapters, scatter plots describe the relationship between two continuous variables. In this section, we'll start with a depiction of a single bivariate relationship (x versus y). We'll then explore ways to enhance this plot by superimposing additional information. Next, we'll learn how to combine several scatter plots into a scatter plot matrix so that you can view many bivariate relationships at once. We'll also review the special case where many data points overlap, limiting our ability to picture the data, and we'll discuss a number of ways around this difficulty. Finally, we'll extend the two-dimensional graph to three dimensions, with the addition of a third continuous variable. This will include 3D scatter plots and bubble plots. Each can help you understand the multivariate relationship among three variables at once.

The basic function for creating a scatter plot in R is `plot(x, y)`, where *x* and *y* are numeric vectors denoting the (*x*, *y*) points to plot. Listing 11.1 presents an example.

Listing 11.1 A scatter plot with best fit lines

```
attach(mtcars)
plot(wt, mpg,
     main="Basic Scatter plot of MPG vs. Weight",
     xlab="Car Weight (lbs/1000)",
     ylab="Miles Per Gallon ", pch=19)

abline(lm(mpg~wt), col="red", lwd=2, lty=1)

lines(lowess(wt,mpg), col="blue", lwd=2, lty=2)
```

The resulting graph is provided in figure 11.1.

The code in listing 11.1 attaches the `mtcars` data frame and creates a basic scatter plot using filled circles for the plotting symbol. As expected, as car weight increases, miles per gallon decreases, though the relationship isn't perfectly linear. The `abline()` function is used to add a linear line of best fit, while the `lowess()` function is used to add a smoothed line. This smoothed line is a nonparametric fit line based on locally weighted polynomial regression. See Cleveland (1981) for details on the algorithm.

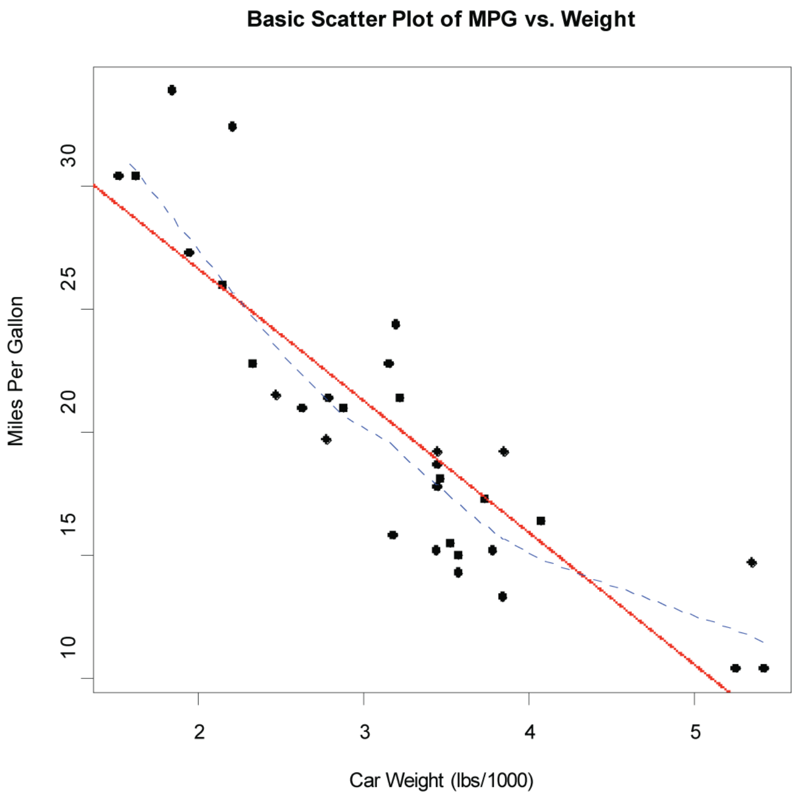


Figure 11.1 Scatter plot of car mileage versus weight, with superimposed linear and lowess fit lines.

NOTE R has two functions for producing lowess fits: `lowess()` and `loess()`. The `loess()` function is a newer, formula-based version of `lowess()` and is more powerful. The two functions have different defaults, so be careful not to confuse them.

The `scatterplot()` function in the `car` package offers many enhanced features and convenience functions for producing scatter plots, including fit lines, marginal box plots, confidence ellipses, plotting by subgroups, and interactive point identification. For example, a more complex version of the previous plot is produced by the following code:

```
library(car)
scatterplot(mpg ~ wt | cyl, data=mtcars, lwd=2,
            main="Scatter Plot of MPG vs. Weight by # Cylinders",
            xlab="Weight of Car (lbs/1000)",
            ylab="Miles Per Gallon",
            legend.plot=TRUE,
            id.method="identify",
            labels=row.names(mtcars),
            boxplots="xy"
)
```

Here, the `scatterplot()` function is used to plot miles per gallon versus weight for automobiles that have four, six, or eight cylinders. The formula `mpg ~ wt | cyl` indicates conditioning (that is, separate plots between `mpg` and `wt` for each level of `cyl`). The graph is provided in figure 11.2.

By default, subgroups are differentiated by color and plotting symbol, and separate linear and loess lines are fit. By default, the loess fit requires five unique data points, so no smoothed fit is plotted for six-cylinder cars. The `id.method` option indicates that points will be identified interactively by mouse clicks, until the user selects Stop (via the Graphics or context-sensitive menu) or the Esc key. The `labels` option indicates that points will be identified with their row names. Here you see that the Toyota Corolla and Fiat 128 have unusually good gas mileage, given their weights. The `legend.plot` option adds a legend to the upper-left margin and marginal box plots

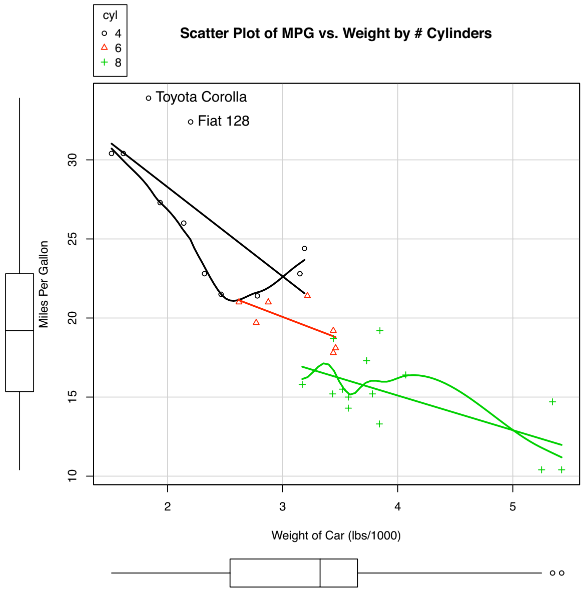


Figure 11.2 Scatter plot with subgroups and separately estimated fit lines

for `mpg` and `weight` are requested with the `boxplots` option. The `scatterplot()` function has many features worth investigating, including robust options and data concentration ellipses not covered here. See `help(scatterplot)` for more details.

Scatter plots help you visualize relationships between quantitative variables, two at a time. But what if you wanted to look at the bivariate relationships between automobile mileage, weight, displacement (cubic inch), and rear axle ratio? One way is to arrange these six scatter plots in a matrix. When there are several quantitative variables, you can represent their relationships in a scatter plot matrix, which is covered next.

11.1.1 Scatter plot matrices

There are at least four useful functions for creating scatter plot matrices in R. Analysts must love scatter plot matrices! A basic scatter plot matrix can be created with the `pairs()` function. The following code produces a scatter plot matrix for the variables `mpg`, `disp`, `drat`, and `wt`:

```
pairs(~mpg+disp+drat+wt, data=mtcars,
      main="Basic Scatter Plot Matrix")
```

All the variables on the right of the ~ are included in the plot. The graph is provided in figure 11.3.

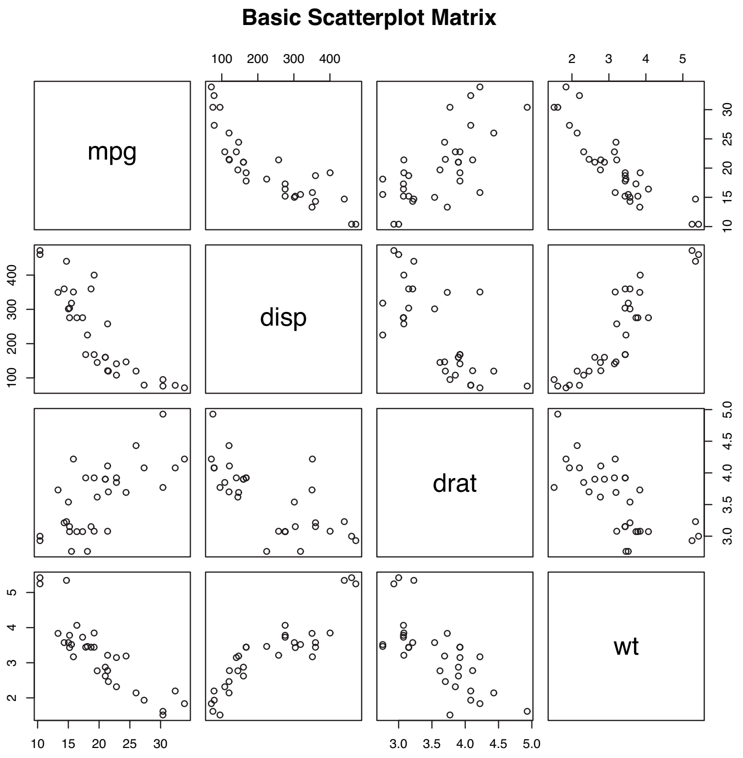


Figure 11.3 Scatter plot matrix created by the `pairs()` function

Here you can see the bivariate relationship among all the variables specified. For example, the scatter plot between mpg and disp is found at the row and column intersection of those two variables. Note that the six scatter plots below the principal diagonal are the same as those above the diagonal. This arrangement is a matter of convenience. By adjusting the options, you could display just the lower or upper triangle. For example, the option upper.panel=NULL would produce a graph with just the lower triangle of plots.

The scatterplotMatrix() function in the car package can also produce scatter plot matrices and can optionally do the following:

- Condition the scatter plot matrix on a factor
- Include linear and loess fit lines
- Place box plots, densities, or histograms in the principal diagonal
- Add rug plots in the margins of the cells

Here's an example:

```
library(car)
scatterplotMatrix(~ mpg + disp + drat + wt, data=mtcars, spread=FALSE,
                  lty.smooth=2, main="Scatter Plot Matrix via car Package")
```

The graph is provided in figure 11.4. Here you can see that linear and smoothed (loess) fit lines are added by default and that kernel density and rug plots are

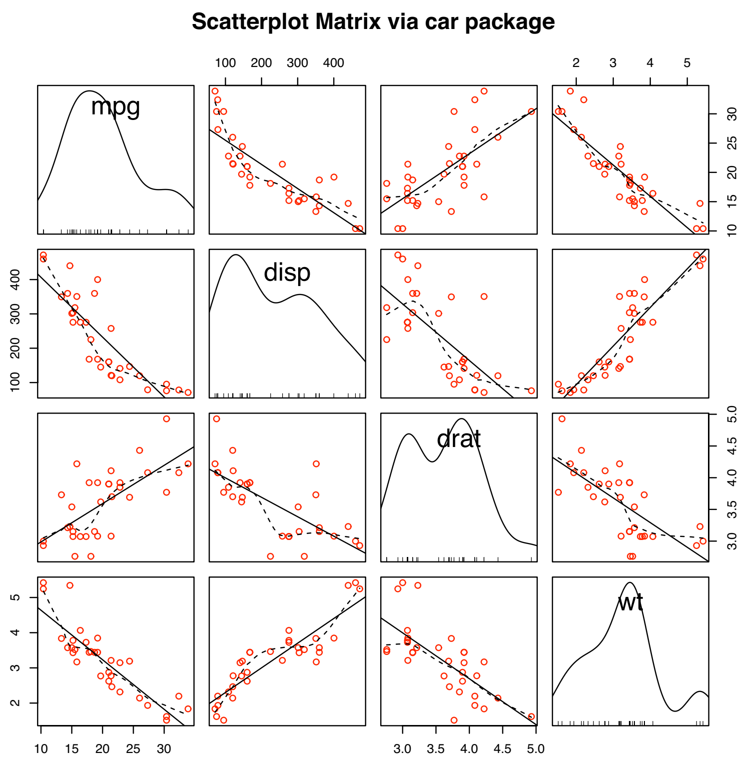


Figure 11.4 **Scatter plot matrix created with the `scatterplotMatrix()` function. The graph includes kernel density and rug plots in the principal diagonal and linear and loess fit lines.**

added to the principal diagonal. The `spread=FALSE` option suppresses lines showing spread and asymmetry, and the `lty.smooth=2` option displays the loess fit lines using dashed rather than solid lines.

As a second example of the `scatterplotMatrix()` function, consider the following code:

```
library(car)
scatterplotMatrix(~ mpg + disp + drat + wt | cyl, data=mtcars,
                  spread=FALSE, diagonal="histogram",
                  main="Scatter Plot Matrix via car Package")
```

Here, you change the kernel density plots to histograms and condition the results on the number of cylinders for each car. The results are displayed in figure 11.5.

By default, the regression lines are fit for the entire sample. Including the option `by.groups = TRUE` would have produced separate fit lines by subgroup.

An interesting variation on the scatter plot matrix is provided by the `cpairs()` function in the `gclus` package. The `cpairs()` function provides options to rearrange

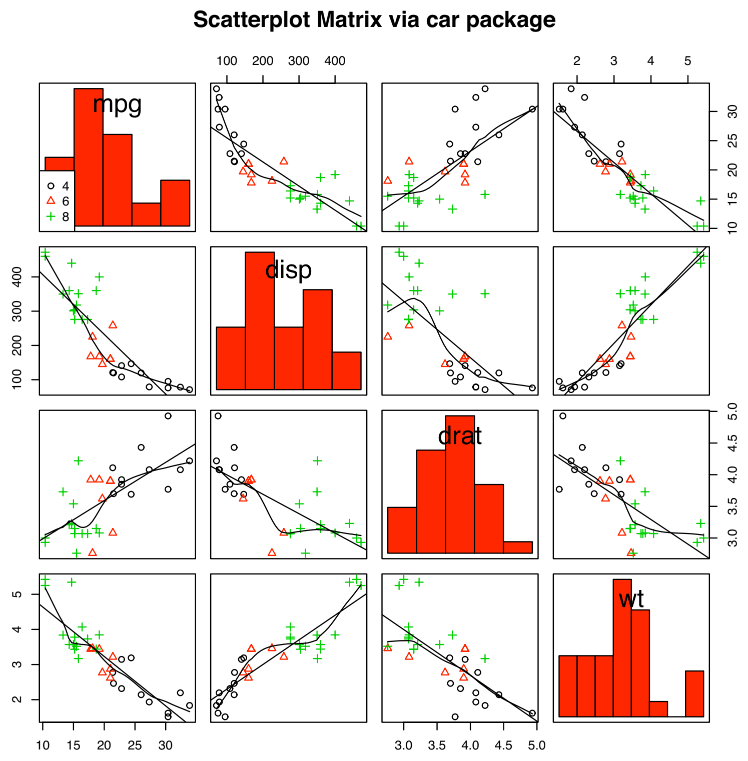


Figure 11.5 Scatter plot matrix produced by the `scatterplot.Matrix()` function. The graph includes histograms in the principal diagonal and linear and loess fit lines. Additionally, subgroups (defined by number of cylinders) are indicated by symbol type and color.

variables in the matrix so that variable pairs with higher correlations are closer to the principal diagonal. The function can also color-code the cells to reflect the size of these correlations. Consider the correlations among `mpg`, `wt`, `disp`, and `drat`:

```
> cor(mtcars[c("mpg", "wt", "disp", "drat")])

        mpg     wt   disp   drat
mpg   1.000 -0.868 -0.848  0.681
wt   -0.868  1.000  0.888 -0.712
disp -0.848  0.888  1.000 -0.710
drat  0.681 -0.712 -0.710  1.000
```

You can see that the highest correlations are between weight and displacement (0.89) and between weight and miles per gallon (–0.87). The lowest correlation is between miles per gallon and rear axle ratio (0.68). You can reorder and color the scatter plot matrix among these variables using the code in the following listing.

Listing 11.2 Scatter plot matrix produced with the `gclus` package

```
library(gclus)
mydata <- mtcars[c(1, 3, 5, 6)]
mydata.corr <- abs(cor(mydata))

mycolors <- dmat.color(mydata.corr)

myorder <- order.single(mydata.corr)

cpairs(mydata,
       myorder,
       panel.colors=mycolors,
       gap=.5,
       main="Variables Ordered and Colored by Correlation"
)
```

The code in listing 11.2 uses the `dmat.color()`, `order.single()`, and `cpairs()` functions from the `gclus` package. First, you select the desired variables from the `mtcars` data frame and calculate the absolute values of the correlations among them. Next, you obtain the colors to plot using the `dmat.color()` function. Given a symmetric matrix (a correlation matrix in this case), `dmat.color()` returns a matrix of colors. You also sort the variables for plotting. The `order.single()` function sorts objects so that similar object pairs are adjacent. In this case, the variable ordering is based on the similarity of the correlations. Finally, the scatter plot matrix is plotted and colored using the new ordering (`myorder`) and the color list (`mycolors`). The `gap` option adds a small space between cells of the matrix. The resulting graph is provided in figure 11.6.

You can see from the figure that the highest correlations are between weight and displacement and weight and miles per gallon (red and closest to the principal diagonal). The lowest correlation is between rear axle ratio and miles per gallon

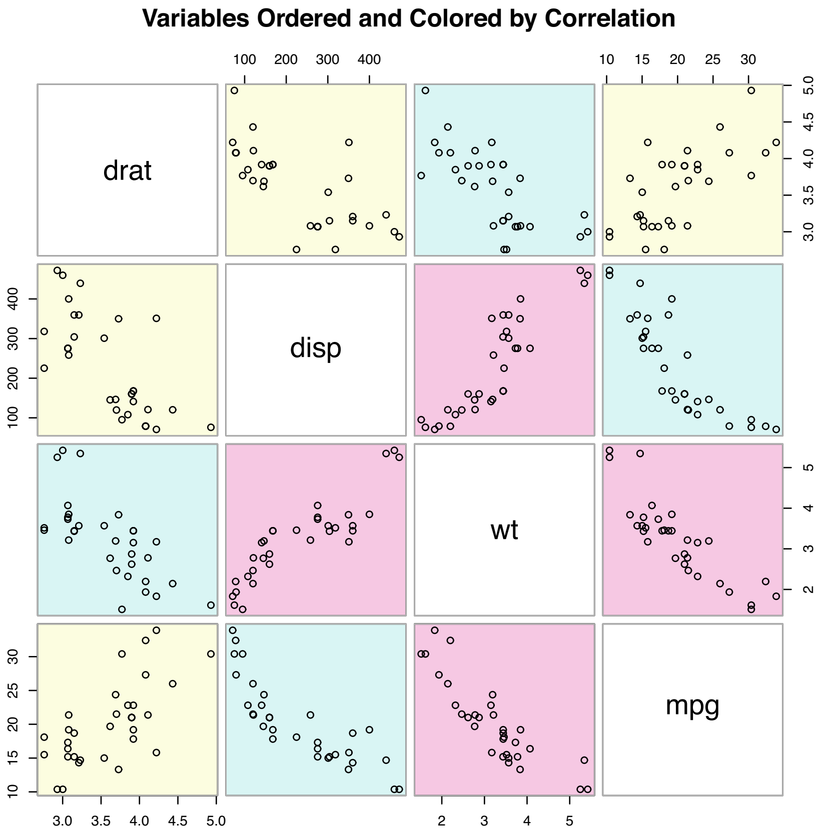


Figure 11.6 Scatter plot matrix produced with the `cpairs()` function in the `gclus` package. Variables closer to the principal diagonal are more highly correlated.

(yellow and far from the principal diagonal). This method is particularly useful when many variables, with widely varying inter-correlations, are considered. You'll see other examples of scatter plot matrices in chapter 16.

11.1.2 High-density scatter plots

When there's a significant overlap among data points, scatter plots become less useful for observing relationships. Consider the following contrived example with 10,000 observations falling into two overlapping clusters of data:

```
set.seed(1234)

n <- 10000
c1 <- matrix(rnorm(n, mean=0, sd=.5), ncol=2)
c2 <- matrix(rnorm(n, mean=3, sd=2), ncol=2)
mydata <- rbind(c1, c2)
mydata <- as.data.frame(mydata)
names(mydata) <- c("x", "y")
```

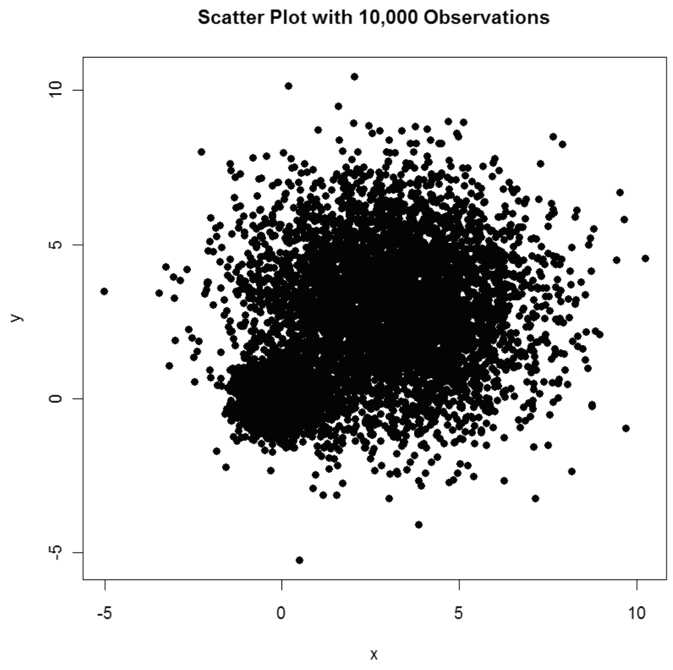


Figure 11.7 **Scatter plot with 10,000 observations and significant overlap of data points. Note that the overlap of data points makes it difficult to discern where the concentration of data is greatest.**

If you generate a standard scatter plot between these variables using the following code

```
with(mydata,
     plot(x, y, pch=19, main="Scatter Plot with 10,000 Observations"))
```

you'll obtain a graph like the one in figure 11.7.

The overlap of data points in figure 11.7 makes it difficult to discern the relationship between `x` and `y`. R provides several graphical approaches that can be used when this occurs. They include the use of binning, color, and transparency to indicate the number of overprinted data points at any point on the graph.

The `smoothScatter()` function uses a kernel density estimate to produce smoothed color density representations of the scatterplot. The following code

```
with(mydata,
     smoothScatter(x, y, main="Scatterplot Colored by Smoothed Densities"))
```

produces the graph in figure 11.8.

Using a different approach, the `hexbin()` function in the `hexbin` package provides bivariate binning into hexagonal cells (it looks better than it sounds). Applying this function to the dataset

```
library(hexbin)
with(mydata, {
     bin <- hexbin(x, y, xbins=50)
     plot(bin, main="Hexagonal Binning with 10,000 Observations")
     })
```

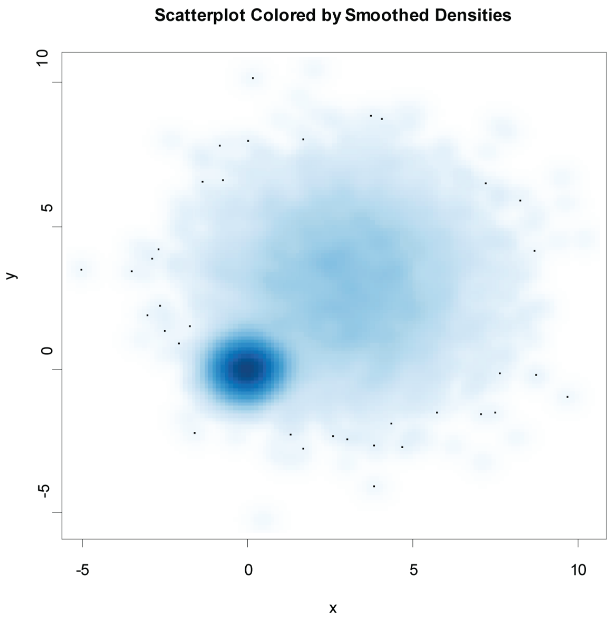


Figure 11.8 Scatterplot using `smoothScatter()` to plot smoothed density estimates. Densities are easy to read from the graph.

you get the scatter plot in figure 11.9.

Finally, the `iplot()` function in the `IDPmisc` package can be used to display density (the number of data points at a specific spot) using color. The code

```
library(IDPmisc)
with(mydata,
     iplot(x, y, main="Image Scatter Plot with Color Indicating Density"))
```

produces the graph in figure 11.10.

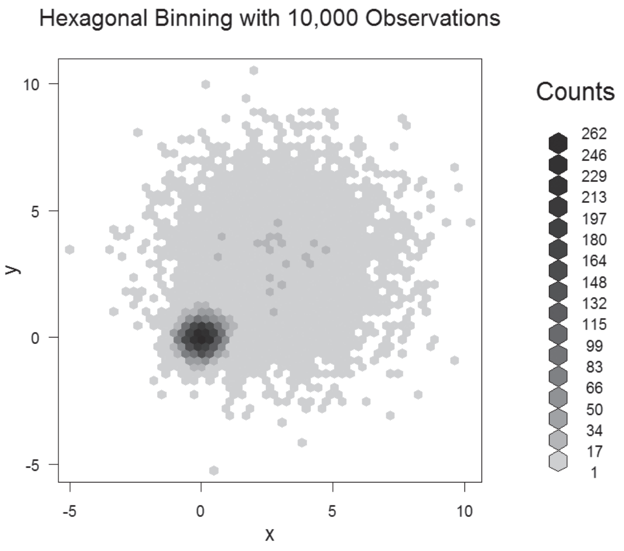


Figure 11.9 Scatter plot using hexagonal binning to display the number of observations at each point. Data concentrations are easy to see and counts can be read from the legend.

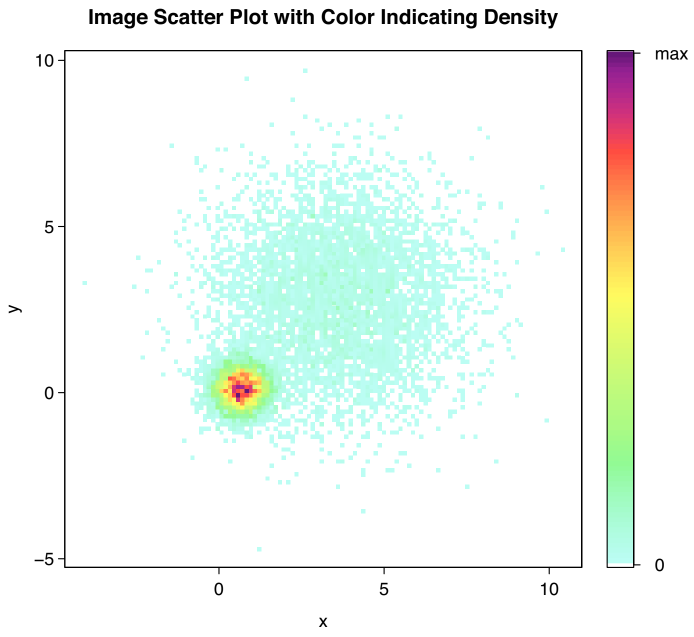


Figure 11.10 Scatter plot of 10,000 observations, where density is indicated by color. The data concentrations are easily discernable.

It's useful to note that the `smoothScatter()` function in the base package, along with the `ipairs()` function in the `IDPmisc` package, can be used to create readable scatter plot matrices for large datasets as well. See `?smoothScatter` and `?ipairs` for examples.

11.1.3 3D scatter plots

Scatter plots and scatter plot matrices display bivariate relationships. What if you want to visualize the interaction of three quantitative variables at once? In this case, you can use a 3D scatter plot.

For example, say that you're interested in the relationship between automobile mileage, weight, and displacement. You can use the `scatterplot3d()` function in the `scatterplot3d` package to picture their relationship. The format is

```
scatterplot3d(x, y, z)
```

where *x* is plotted on the horizontal axis, *y* is plotted on the vertical axis, and *z* is plotted in perspective. Continuing our example

```
library(scatterplot3d)
attach(mtcars)
scatterplot3d(wt, disp, mpg,
    main="Basic 3D Scatter Plot")
```

produces the 3D scatter plot in figure 11.11.

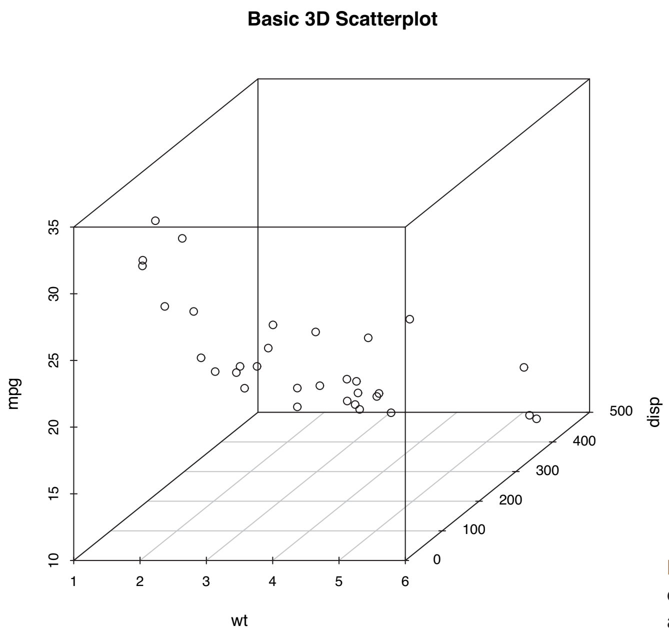


Figure 11.11 3D scatter plot of miles per gallon, auto weight, and displacement

The `scatterplot3d()` function offers many options, including the ability to specify symbols, axes, colors, lines, grids, highlighting, and angles. For example, the code

```
library(scatterplot3d)
attach(mtcars)
scatterplot3d(wt, disp, mpg,
              pch=16,
              highlight.3d=TRUE,
              type="h",
              main="3D Scatter Plot with Vertical Lines")
```

produces a 3D scatter plot with highlighting to enhance the impression of depth, and vertical lines connecting points to the horizontal plane (see figure 11.12).

As a final example, let's take the previous graph and add a regression plane. The necessary code is:

```
library(scatterplot3d)
attach(mtcars)
s3d <-scatterplot3d(wt, disp, mpg,
      pch=16,
      highlight.3d=TRUE,
      type="h",
      main="3D Scatter Plot with Vertical Lines and Regression Plane")
fit <- lm(mpg ~ wt+disp)
s3d$plane3d(fit)
```

The resulting graph is provided in figure 11.13.

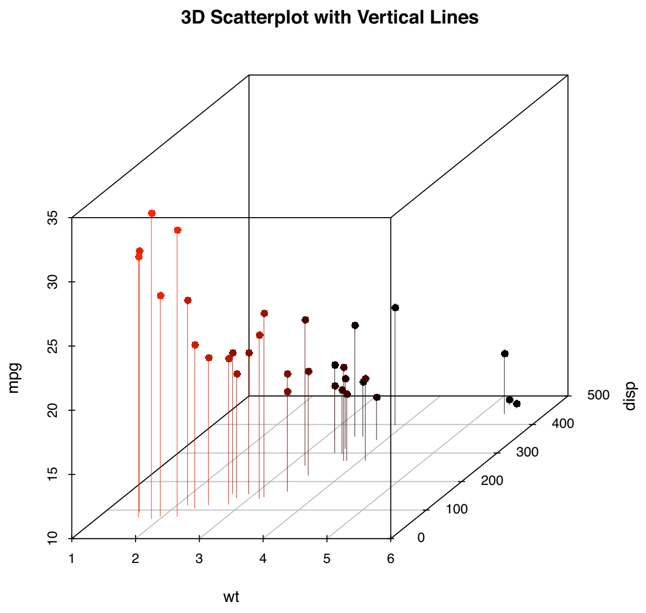


Figure 11.12 3D scatter plot with vertical lines and shading

The graph allows you to visualize the prediction of miles per gallon from automobile weight and displacement using a multiple regression equation. The plane represents the predicted values, and the points are the actual values. The vertical distances from the plane to the points are the residuals. Points that lie above the plane are under-predicted, while points that lie below the line are over-predicted. Multiple regression is covered in chapter 8.

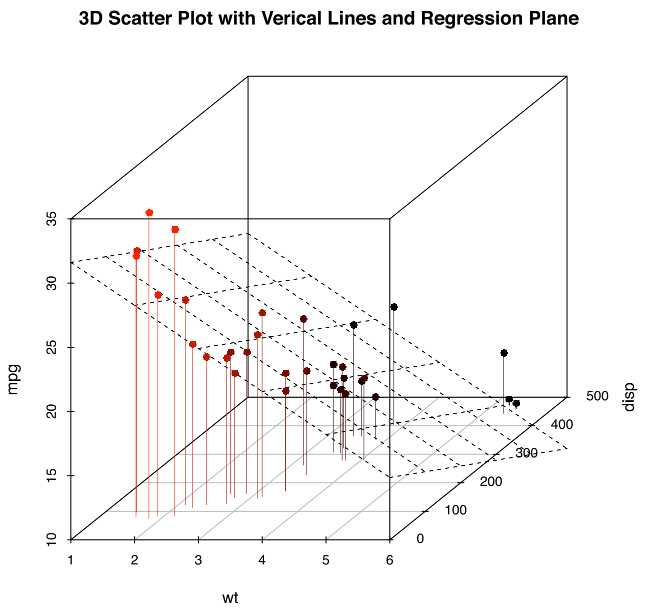


Figure 11.13 3D scatter plot with vertical lines, shading, and overlaid regression plane

SPINNING 3D SCATTER PLOTS

Three-dimensional scatter plots are much easier to interpret if you can interact with them. R provides several mechanisms for rotating graphs so that you can see the plotted points from more than one angle.

For example, you can create an interactive 3D scatter plot using the `plot3d()` function in the `rgl` package. It creates a spinning 3D scatter plot that can be rotated with the mouse. The format is

```
plot3d(x, y, z)
```

where *x*, *y*, and *z* are numeric vectors representing points. You can also add options like `col` and `size` to control the color and size of the points, respectively. Continuing our example, try the code

```
library(rgl)
attach(mtcars)
plot3d(wt, disp, mpg, col="red", size=5)
```

You should get a graph like the one depicted in figure 11.14. Use the mouse to rotate the axes. I think that you'll find that being able to rotate the scatter plot in three dimensions makes the graph much easier to understand.

You can perform a similar function with the `scatter3d()` in the `Rcmdr` package:

```
library(Rcmdr)
attach(mtcars)
scatter3d(wt, disp, mpg)
```

The results are displayed in figure 11.15.

The `scatter3d()` function can include a variety of regression surfaces, such as linear, quadratic, smooth, and additive. The linear surface depicted is the default. Additionally, there are options for interactively identifying points. See `help(scatter3d)` for more details. I'll have more to say about the `Rcmdr` package in appendix A.

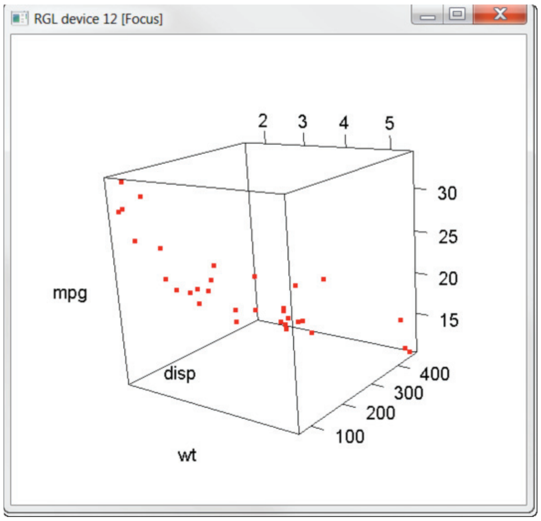


Figure 11.14 Rotating 3D scatter plot produced by the `plot3d()` function in the `rgl` package

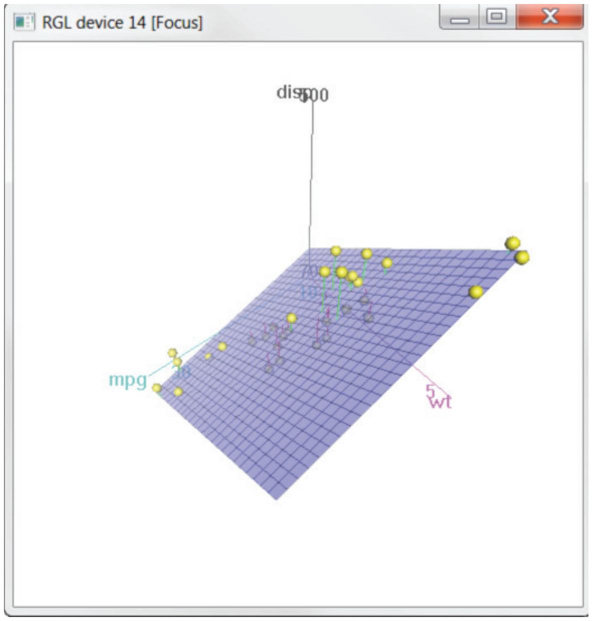


Figure 11.15 **Spinning 3D scatter plot produced by the `scatter3d()` function in the `Rcmdr` package**

11.1.4 Bubble plots

In the previous section, you displayed the relationship between three quantitative variables using a 3D scatter plot. Another approach is to create a 2D scatter plot and use the size of the plotted point to represent the value of the third variable. This approach is referred to as a *bubble plot*.

You can create a bubble plot using the `symbols()` function. This function can be used to draw circles, squares, stars, thermometers, and box plots at a specified set of (*x*, *y*) coordinates. For plotting circles, the format is

```
symbols(x, y, circle=radius)
```

where *x* and *y* and *radius* are vectors specifying the x and y coordinates and circle radiuses, respectively.

You want the areas, rather than the radiuses of the circles, to be proportional to the values of a third variable. Given the formula for the radius of a circle ($r = \sqrt{\frac{A}{\pi}}$) the proper call is

```
symbols(x, y, circle=sqrt(z/pi))
```

where *z* is the third variable to be plotted.

Let's apply this to the `mtcars` data, plotting car weight on the x-axis, miles per gallon on the y-axis, and engine displacement as the bubble size. The following code

```
attach(mtcars)
r <- sqrt(disp/pi)
symbols(wt, mpg, circle=r, inches=0.30,
        fg="white", bg="lightblue",
        main="Bubble Plot with point size proportional to displacement",
        ylab="Miles Per Gallon",
        xlab="Weight of Car (lbs/1000)")
text(wt, mpg, rownames(mtcars), cex=0.6)
detach(mtcars)
```

produces the graph in figure 11.16. The option `inches` is a scaling factor that can be used to control the size of the circles (the default is to make the largest circle 1 inch). The `text()` function is optional. Here it is used to add the names of the cars to the plot. From the figure, you can see that increased gas mileage is associated with both decreased car weight and engine displacement.

In general, statisticians involved in the R project tend to avoid bubble plots for the same reason they avoid pie charts. Humans typically have a harder time making judgments about volume than distance. But bubble charts are certainly popular in the business world, so I'm including them here for completeness.

I've certainly had a lot to say about scatter plots. This attention to detail is due, in part, to the central place that scatter plots hold in data analysis. While simple, they can help you visualize your data in an immediate and straightforward manner, uncovering relationships that might otherwise be missed.

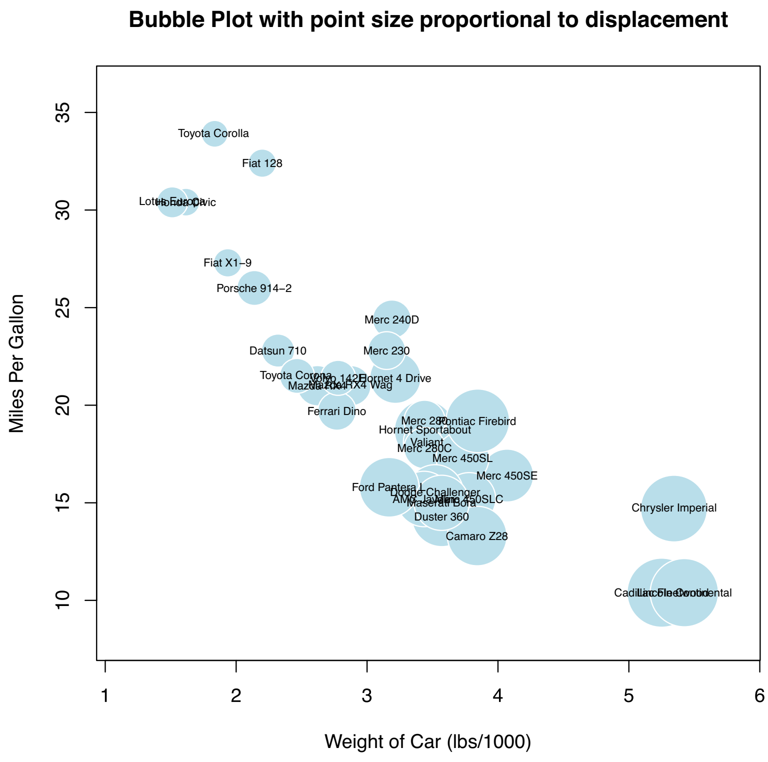


Figure 11.16 Bubble plot of car weight versus `mpg` where point size is proportional to engine displacement

11.2 Line charts

If you connect the points in a scatter plot moving from left to right, you have a line plot. The dataset `Orange` that come with the base installation contains age and circumference data for five orange trees. Consider the growth of the first orange tree, depicted in figure 11.17. The plot on the left is a scatter plot, and the plot on the right is a line chart. As you can see, line charts are particularly good vehicles for conveying change.

The graphs in figure 11.17 were created with the code in the following listing.

Listing 11.3 Creating side-by-side scatter and line plots

```
opar <- par(no.readonly=TRUE)
par(mfrow=c(1,2))
t1 <- subset(Orange, Tree==1)
plot(t1$age, t1$circumference,
     xlab="Age (days)",
     ylab="Circumference (mm)",
     main="Orange Tree 1 Growth")
plot(t1$age, t1$circumference,
     xlab="Age (days)",
     ylab="Circumference (mm)",
     main="Orange Tree 1 Growth",
     type="b")
par(opar)
```

You've seen the elements that make up this code in chapter 3, so I won't go into details here. The main difference between the two plots in figure 11.17 is produced by the option `type="b"`. In general, line charts are created with one of the following two functions

```
plot(x, y, type=)
lines(x, y, type=)
```

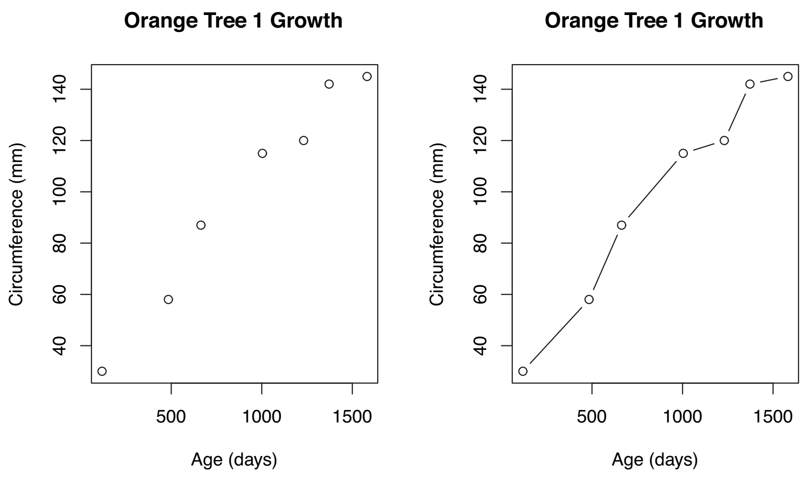


Figure 11.17 Comparison of a scatter plot and a line plot

where x and y are numeric vectors of (x,y) points to connect. The option `type=` can take the values described in table 11.1.

Table 11.1 Line chart options

Type	What is plotted
`p`	Points only
`l`	Lines only
`o`	Over-plotted points (that is, lines overlaid on top of points)
`b, c`	Points (empty if `c`) joined by lines
`s, S`	Stair steps
`h`	Histogram-line vertical lines
`n`	Doesn't produce any points or lines (used to set up the axes for later commands)

Examples of each type are given in figure 11.18. As you can see, `type="p"` produces the typical scatter plot. The option `type="b"` is the most common for line charts. The difference between `b` and `c` is whether the points appear or gaps are left instead. Both `type="s"` and `type="S"` produce stair steps (step functions). The first runs, then rises, whereas the second rises, then runs.

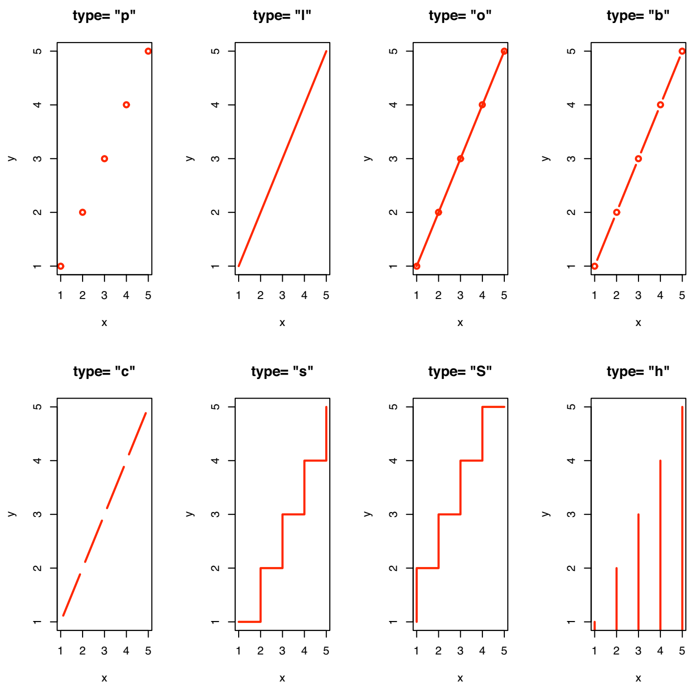


Figure 11.18 `type=` **options in the** `plot()` **and** `lines()` **functions**

There's an important difference between the `plot()` and `lines()` functions. The `plot()` function will create a new graph when invoked. The `lines()` function *adds* information to an existing graph but *can't* produce a graph on its own.

Because of this, the `lines()` function is typically used after a `plot()` command has produced a graph. If desired, you can use the `type="n"` option in the `plot()` function to set up the axes, titles, and other graph features, and then use the `lines()` function to add various lines to the plot.

To demonstrate the creation of a more complex line chart, let's plot the growth of all five orange trees over time. Each tree will have its own distinctive line. The code is shown in the next listing and the results in figure 11.19.

Listing 11.4 Line chart displaying the growth of five orange trees over time

```
Orange$Tree <- as.numeric(Orange$Tree)
ntrees <- max(Orange$Tree)

xrange <- range(Orange$age)
yrange <- range(Orange$circumference)

plot(xrange, yrange,
     type="n",
     xlab="Age (days)",
     ylab="Circumference (mm)"
 )

colors <- rainbow(ntrees)
linetype <- c(1:ntrees)
plotchar <- seq(18, 18+ntrees, 1)

for (i in 1:ntrees) {
    tree <- subset(Orange, Tree==i)
    lines(tree$age, tree$circumference,
          type="b",
          lwd=2,
          lty=linetype[i],
          col=colors[i],
          pch=plotchar[i]
    )
}

title("Tree Growth", "example of line plot")

legend(xrange[1], yrange[2],
       1:ntrees,
       cex=0.8,
       col=colors,
       pch=plotchar,
       lty=linetype,
       title="Tree"
)
```

Convert factor to numeric for convenience

Set up plot

Add lines

Add legend

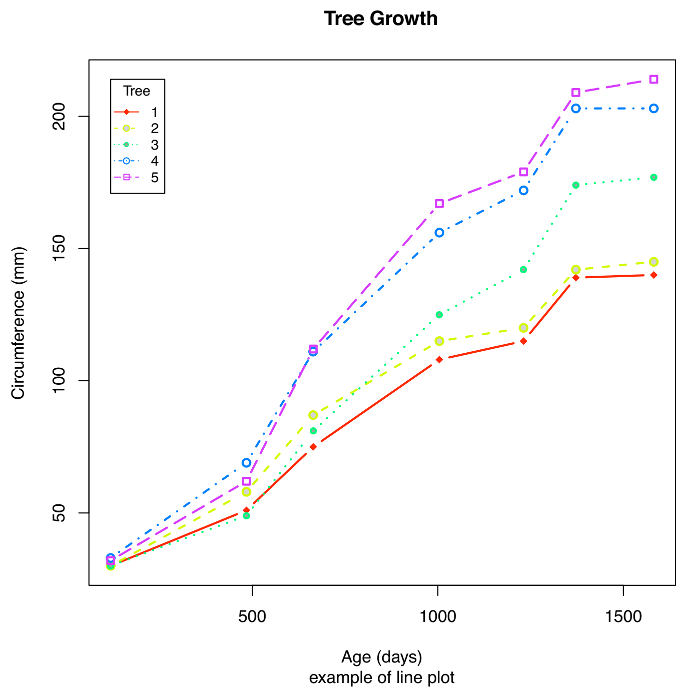


Figure 11.19 Line chart displaying the growth of five orange trees

In listing 11.4, the `plot()` function is used to set up the graph and specify the axis labels and ranges but plots no actual data. The `lines()` function is then used to add a separate line and set of points for each orange tree. You can see that tree 4 and tree 5 demonstrated the greatest growth across the range of days measured, and that tree 5 overtakes tree 4 at around 664 days.

Many of the programming conventions in R that I discussed in chapters 2, 3, and 4 are used in listing 11.4. You may want to test your understanding by working through each line of code and visualizing what it's doing. If you can, you are on your way to becoming a serious R programmer (and fame and fortune is near at hand)! In the next section, you'll explore ways of examining a number of correlation coefficients at once.

11.3 Correlograms

Correlation matrices are a fundamental aspect of multivariate statistics. Which variables under consideration are strongly related to each other and which aren't? Are there clusters of variables that relate in specific ways? As the number of variables grow, such questions can be harder to answer. Correlograms are a relatively recent tool for visualizing the data in correlation matrices.

It's easier to explain a correlogram once you've seen one. Consider the correlations among the variables in the `mtcars` data frame. Here you have 11 variables, each measuring some aspect of 32 automobiles. You can get the correlations using the following code:

```
> options(digits=2)
> cor(mtcars)
```

```
       mpg   cyl  disp    hp   drat    wt   qsec    vs     am  gear   carb
mpg   1.00 -0.85 -0.85 -0.78  0.681 -0.87  0.419  0.66  0.600  0.48 -0.551
cyl  -0.85  1.00  0.90  0.83 -0.700  0.78 -0.591 -0.81 -0.523 -0.49  0.527
disp -0.85  0.90  1.00  0.79 -0.710  0.89 -0.434 -0.71 -0.591 -0.56  0.395
hp   -0.78  0.83  0.79  1.00 -0.449  0.66 -0.708 -0.72 -0.243 -0.13  0.750
drat  0.68 -0.70 -0.71 -0.45  1.000 -0.71  0.091  0.44  0.713  0.70 -0.091
wt   -0.87  0.78  0.89  0.66 -0.712  1.00 -0.175 -0.55 -0.692 -0.58  0.428
qsec  0.42 -0.59 -0.43 -0.71  0.091 -0.17  1.000  0.74 -0.230 -0.21 -0.656
vs    0.66 -0.81 -0.71 -0.72  0.440 -0.55  0.745  1.00  0.168  0.21 -0.570
am    0.60 -0.52 -0.59 -0.24  0.713 -0.69 -0.230  0.17  1.000  0.79  0.058
gear  0.48 -0.49 -0.56 -0.13  0.700 -0.58 -0.213  0.21  0.794  1.00  0.274
carb -0.55  0.53  0.39  0.75 -0.091  0.43 -0.656 -0.57  0.058  0.27  1.000
```

Which variables are most related? Which variables are relatively independent? Are there any patterns? It isn't that easy to tell from the correlation matrix without significant time and effort (and probably a set of colored pens to make notations).

You can display that same correlation matrix using the `corrgram()` function in the `corrgram` package (see figure 11.20). The code is:

```
library(corrgram)
corrgram(mtcars, order=TRUE, lower.panel=panel.shade,
         upper.panel=panel.pie, text.panel=panel.txt,
         main="Correlogram of mtcars intercorrelations")
```

To interpret this graph, start with the lower triangle of cells (the cells below the principal diagonal). By default, a blue color and hashing that goes from lower left to upper right represents a positive correlation between the two variables that meet at that cell. Conversely, a red color and hashing that goes from the upper left to the lower right represents a negative correlation. The darker and more saturated the color, the greater the magnitude of the correlation. Weak correlations, near zero, will appear washed out. In the current graph, the rows and columns have been reordered (using principal components analysis) to cluster variables together that have similar correlation patterns.

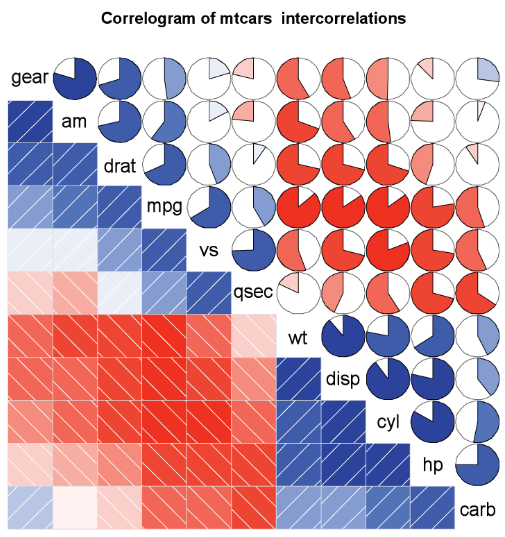


Figure 11.20 Correlogram of the correlations among the variables in the `mtcars` data frame. Rows and columns have been reordered using principal components analysis.

You can see from shaded cells that `gear`, `am`, `drat`, and `mpg` are positively correlated with one another. You can also see that `wt`, `disp`, `cyl`, `hp`, and `carb` are positively correlated with one another. But the first group of variables is negatively correlated with the second group of variables. You can also see that the correlation between `carb` and `am` is weak, as is the correlation between `vs` and `gear`, `vs` and `am`, and `drat` and `qsec`.

The upper triangle of cells displays the same information using pies. Here, color plays the same role, but the strength of the correlation is displayed by the size of the filled pie slice. Positive correlations fill the pie starting at 12 o'clock and moving in a clockwise direction. Negative correlations fill the pie by moving in a counterclockwise direction. The format of the `corrgram()` function is

```
corrgram(x, order=, panel=, text.panel=, diag.panel=)
```

where *x* is a data frame with one observation per row. When `order=TRUE`, the variables are reordered using a principal component analysis of the correlation matrix. Reordering can help make patterns of bivariate relationships more obvious.

The option `panel` specifies the type of off-diagonal panels to use. Alternatively, you can use the options `lower.panel` and `upper.panel` to choose different options below and above the main diagonal. The `text.panel` and `diag.panel` options refer to the main diagonal. Allowable values for `panel` are described in table 11.2.

Table 11.2 Panel options for the `corrgram()` function

Placement	Panel Option	Description
Off diagonal	`panel.pie`	The filled portion of the pie indicates the magnitude of the correlation.
	`panel.shade`	The depth of the shading indicates the magnitude of the correlation.
	`panel.ellipse`	A confidence ellipse and smoothed line are plotted.
	`panel.pts`	A scatter plot is plotted.
Main diagonal	`panel.minmax`	The minimum and maximum values of the variable are printed.
	`panel.txt`	The variable name is printed.

Let's try a second example. The code

```
library(corrgram)
corrgram(mtcars, order=TRUE, lower.panel=panel.ellipse,
         upper.panel=panel.pts, text.panel=panel.txt,
         diag.panel=panel.minmax,
         main="Correlogram of mtcars data using scatter plots and ellipses")
```

produces the graph in figure 11.21. Here you're using smoothed fit lines and confidence ellipses in the lower triangle and scatter plots in the upper triangle.

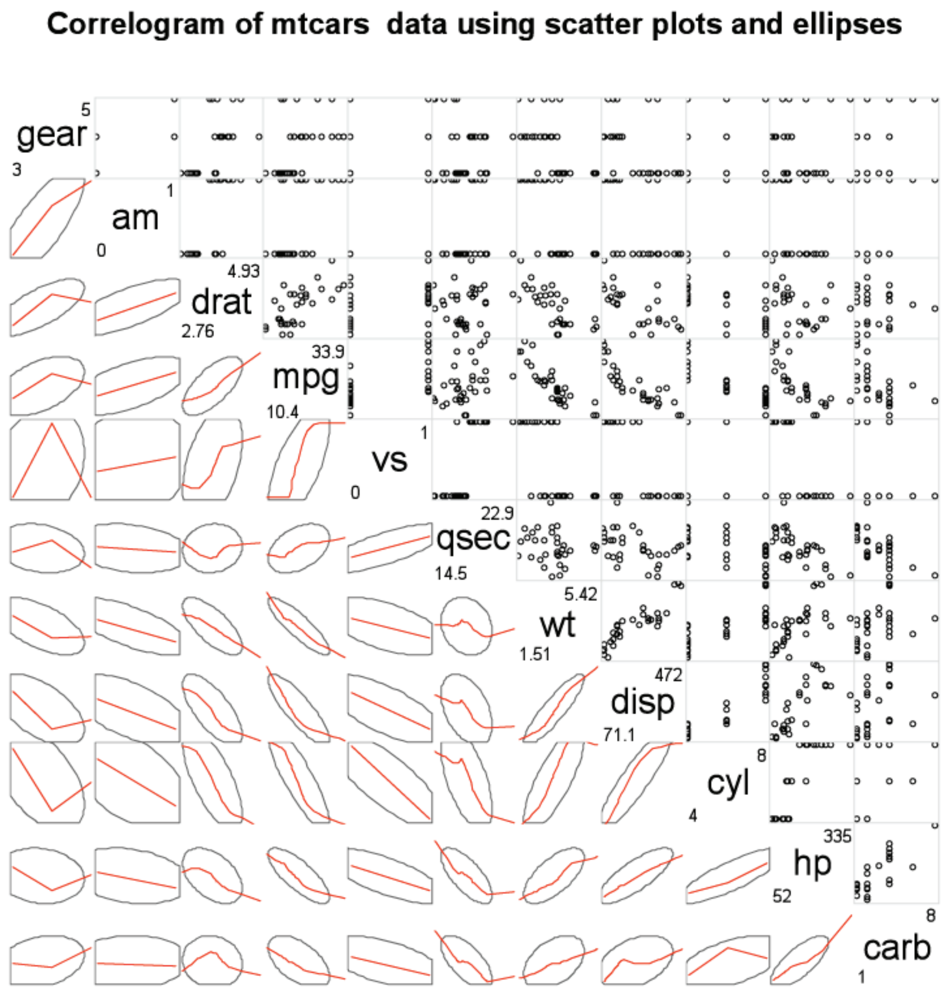


Figure 11.21 Correlogram of the correlations among the variables in the `mtcars` data frame. The lower triangle contains smoothed best fit lines and confidence ellipses, and the upper triangle contains scatter plots. The diagonal panel contains minimum and maximum values. Rows and columns have been reordered using principal components analysis.

Why do the scatter plots look odd?

Several of the variables that are plotted in figure 11.21 have limited allowable values. For example, the number of gears is 3, 4, or 5. The number of cylinders is 4, 6, or 8. Both am (transmission type) and vs (V/S) are dichotomous. This explains the odd-looking scatter plots in the upper diagonal.

Always be careful that the statistical methods you choose are appropriate to the form of the data. Specifying these variables as ordered or unordered factors can serve as a useful check. When R knows that a variable is categorical or ordinal, it attempts to apply statistical methods that are appropriate to that level of measurement.

We'll finish with one more example. The code

```
library(corrgram)
corrgram(mtcars, lower.panel=panel.shade,
         upper.panel=NULL, text.panel=panel.txt,
         main="Car Mileage Data (unsorted)")
```

produces the graph in figure 11.22. Here we're using shading in the lower triangle, keeping the original variable order, and leaving the upper triangle blank.

Before moving on, I should point out that you can control the colors used by the `corrgram()` function. To do so, specify four colors in the `colorRampPalette()` function within the `col.corrgram()` function. Here's an example:

```
library(corrgram)
col.corrgram <- function(ncol){
                colorRampPalette(c("darkgoldenrod4", "burlywood1",
                                   "darkkhaki", "darkgreen"))(ncol)}
corrgram(mtcars, order=TRUE, lower.panel=panel.shade,
         upper.panel=panel.pie, text.panel=panel.txt,
         main="A Corrgram (or Horse) of a Different Color")
```

Try it and see what you get.

Correlograms can be a useful way to examine large numbers of bivariate relationships among quantitative variables. Because they're relatively new, the greatest challenge is to educate the recipient on how to interpret them.

To learn more, see Michael Friendly's article "*Corrgrams: Exploratory Displays for Correlation Matrices,*" available at http://www.math.yorku.ca/SCS/Papers/corrgram.pdf.

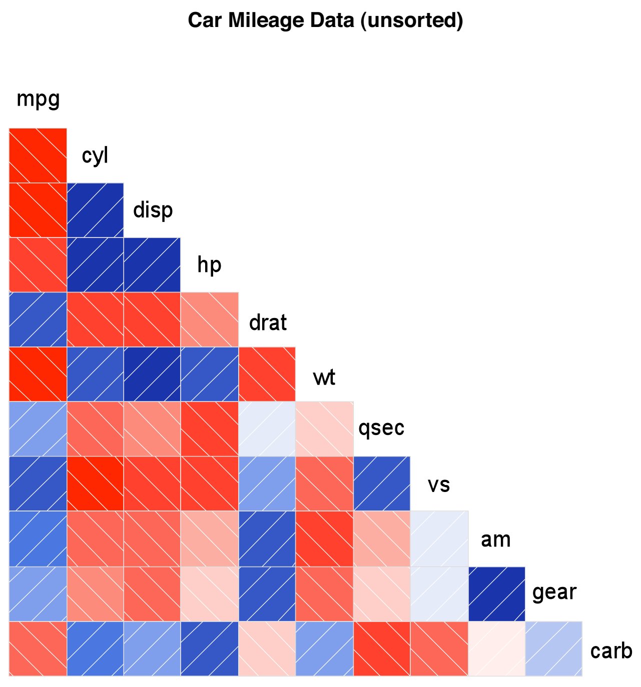


Figure 11.22 Correlogram of the correlations among the variables in the `mtcars` data frame. The lower triangle is shaded to represent the magnitude and direction of the correlations. The variables are plotted in their original order.

11.4 Mosaic plots

Up to this point, we've been exploring methods of visualizing relationships among quantitative/continuous variables. But what if your variables are categorical? When you're looking at a single categorical variable, you can use a bar or pie chart. If there are two categorical variables, you can look at a 3D bar chart (which, by the way, is not so easy to do in R). But what do you do if there are more than two categorical variables?

One approach is to use mosaic plots. In a mosaic plot, the frequencies in a multidimensional contingency table are represented by nested rectangular regions that are proportional to their cell frequency. Color and or shading can be used to represent residuals from a fitted model. For details, see Meyer, Zeileis and Hornick (2006), or Michael Friendly's Statistical Graphics page (http://datavis.ca). Steve Simon has created a good conceptual tutorial on how mosaic plots are created, available at http://www.childrensmercy.org/stats/definitions/mosaic.htm.

Mosaic plots can be created with the `mosaic()` function from the `vcd` library (there's a `mosaicplot()` function in the basic installation of R, but I recommend you use the `vcd` package for its more extensive features). As an example, consider the Titanic dataset available in the base installation. It describes the number of passengers who survived or died, cross-classified by their class (1st, 2nd, 3rd, Crew), sex (Male, Female), and age (Child, Adult). This is a well-studied dataset. You can see the cross-classification using the following code:

```
> ftable(Titanic)
                   Survived  No Yes
Class Sex    Age
1st   Male   Child            0   5
             Adult          118  57
      Female Child            0   1
             Adult            4 140
2nd   Male   Child            0  11
             Adult          154  14
      Female Child            0  13
             Adult           13  80
3rd   Male   Child           35  13
             Adult          387  75
      Female Child           17  14
             Adult           89  76
Crew  Male   Child            0   0
             Adult          670 192
      Female Child            0   0
             Adult            3  20
```

The `mosaic()` function can be invoked as

```
mosaic(table)
```

where *`table`* is a contingency table in array form, or

```
mosaic(formula, data=)
```

where *`formula`* is a standard R formula, and data specifies either a data frame or table. Adding the option `shade=TRUE` will color the figure based on Pearson residuals from a fitted model (independence by default) and the option `legend=TRUE` will display a legend for these residuals.

For example, both

```
library(vcd)
mosaic(Titanic, shade=TRUE, legend=TRUE)
```

and

```
library(vcd)
mosaic(~Class+Sex+Age+Survived, data=Titanic, shade=TRUE, legend=TRUE)
```

will produce the graph shown in figure 11.23. The formula version gives you greater control over the selection and placement of variables in the graph.

There's a great deal of information packed into this one picture. For example, as one moves from crew to first class, the survival rate increases precipitously. Most children were in third and second class. Most females in first class survived, whereas only about half the females in third class survived. There were few females in the crew, causing the Survived labels (No, Yes at the bottom of the chart) to overlap for this group. Keep looking and you'll see many more interesting facts. Remember to look at the relative widths and heights of the rectangles. What else can you learn about that night?

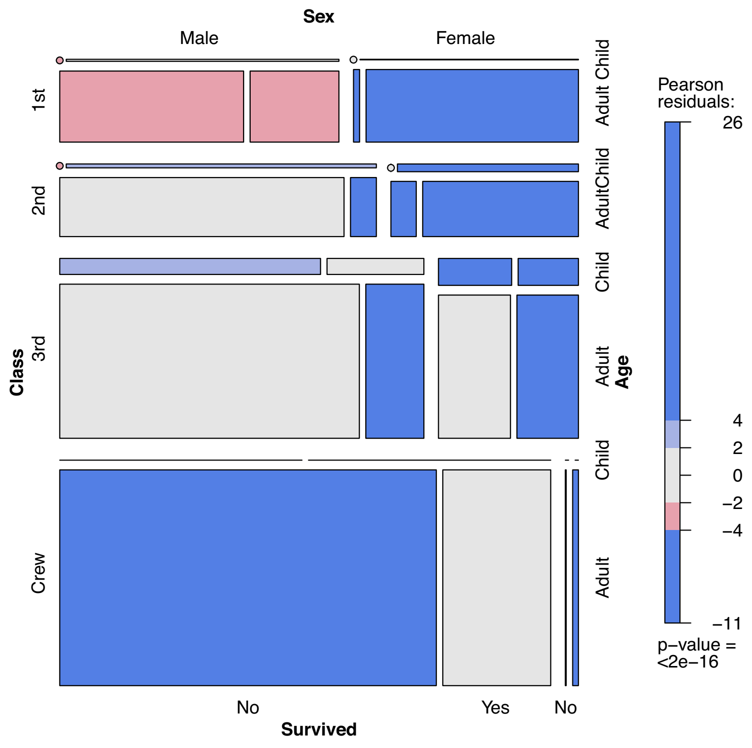


Figure 11.23 **Mosaic plot describing Titanic survivors by class, sex, and age**

Extended mosaic plots add color and shading to represent the residuals from a fitted model. In this example, the blue shading indicates cross-classifications that occur more often than expected, assuming that survival is unrelated to class, gender, and age. Red shading indicates cross-classifications that occur less often than expected under the independence model. Be sure to run the example so that you can see the results in color. The graph indicates that more first-class women survived and more male crew members died than would be expected under an independence model. Fewer third-class men survived than would be expected if survival was independent of class, gender, and age. If you would like to explore mosaic plots in greater detail, try running `example(mosaic)`.

11.5 Summary

In this chapter, we considered a wide range of techniques for displaying relationships among two or more variables. This included the use of 2D and 3D scatter plots, scatter plot matrices, bubble plots, line plots, correlograms, and mosaic plots. Some of these methods are standard techniques, while some are relatively new.

Taken together with methods that allow you to customize graphs (chapter 3), display univariate distributions (chapter 6), explore regression models (chapter 8), and visualize group differences (chapter 9), you now have a comprehensive toolbox for visualizing and extracting meaning from your data.

In later chapters, you'll expand your skills with additional specialized techniques, including graphics for latent variable models (chapter 14), methods for visualizing missing data patterns (chapter 15), and techniques for creating graphs that are conditioned on one or more variables (chapter 16).

In the next chapter, we'll explore resampling statistics and bootstrapping. These are computer intensive methods that allow you to analyze data in new and unique ways.