Chapter 7
Graphing Tools

Chapter 5, the `plot` function was introduced. We demonstrated elementary scatterplots, modifying plotting characters, and adding *x*- and *y*-labels and a main title. In this chapter, we introduce more graphing tools. Not all of them are among our favourites. For example, we have never used pie charts or bar charts. However, these graphs seem to be on the shortlist of so many scientists that we find it necessary to include them in this book. They are discussed in Sections 7.1 and 7.2. Tools to detect outliers—the boxplot and Cleveland dotplot—are presented in Sections 7.3 and 7.4, respectively. We also demonstrate graphs illustrating the mean with lines added to represent the standard error. Scatterplots are further discussed in Section 7.5. Multipanel scatterplots are discussed in Sections 7.6 and 7.7, and advanced tools to display multiple graphs in a single window are presented in Section 7.8.

7.1 The Pie Chart

7.1.1 Pie Chart Showing Avian Influenza Data

We demonstrate the pie chart using the avian influenza dataset from Exercise 1 in Section 3.7. Recall that the data represent the numbers of confirmed human cases of Avian Influenza A/(H5N1) reported to the World Health Organization (WHO). The data for several countries were taken from the WHO website at www.who.int and are reproduced only for educational purposes. We exported the data in the Excel file, *BidFlu.xls*, to a tab-separated ascii file with the name *Birdflucases.txt*. The following code imports the data and presents the usual information.

```
> setwd("C:/RBook/")
> BFCases <- read.table(file = "Birdflucases.txt",
                        header = TRUE)
> names(BFCases)
 [1] "Year"      "Azerbaijan"  "Bangladesh"
 [4] "Cambodia"  "China"       "Djibouti"
 [7] "Egypt"     "Indonesia."  "Iraq"
[10] "LaoPDR"    "Myanmar"     "Nigeria"
[13] "Pakistan"  "Thailand"    "Turkey"
[16] "VietNam"

> str(BFCases)

'data.frame': 6 obs. of 16 variables:
$Year      : int 2003 2004 2005 2006 2007 2008
$Azerbaijan: int 0 0 0 8 0 0
$Bangladesh: int 0 0 0 0 0 1
$Cambodia  : int 0 0 4 2 1 0
$China     : int 1 0 8 13 5 3
$Djibouti  : int 0 0 0 1 0 0
$Egypt     : int 0 0 0 18 25 7
$Indonesia.: int 0 0 20 55 42 18
$Iraq      : int 0 0 0 3 0 0
$LaoPDR    : int 0 0 0 0 2 0
$Myanmar   : int 0 0 0 0 1 0
$Nigeria   : int 0 0 0 0 1 0
$Pakistan  : int 0 0 0 0 3 0
$Thailand  : int 0 17 5 3 0 0
$Turkey    : int 0 0 0 12 0 0
$VietNam   : int 3 29 61 0 8 5
```

We have annual data from the years 2003–2008. The first variable contains the years. There are various things we can learn from this dataset. An interesting question is whether the number of bird flu cases has increased over time. We can address this question for individual countries or for the total number of cases. The latter is calculated by

```
> Cases <- rowSums(BFCases[, 2:16])
> names(Cases) <- BFCases[, 1]
> Cases

2003 2004 2005 2006 2007 2008
 4   46   98   115   88   34
```

Columns 2–16 of `BFCases` contain the information per country. The `rowSums` function calculates totals per year and the `names` function adds the labels 2003–2008 to the variable `Cases`. (Note that the 34 cases in 2008 is misleading, as this was written halfway through 2008. If this were a proper statistical analysis, the 2008 data would be dropped.) The function for a pie chart in R is `pie`. It has various options, some of which are illustrated in Fig. 7.1. The `pie` function requires as input a vector of nonnegative numerical quantities; anything more is optional and deals with labels, colours, and the like.

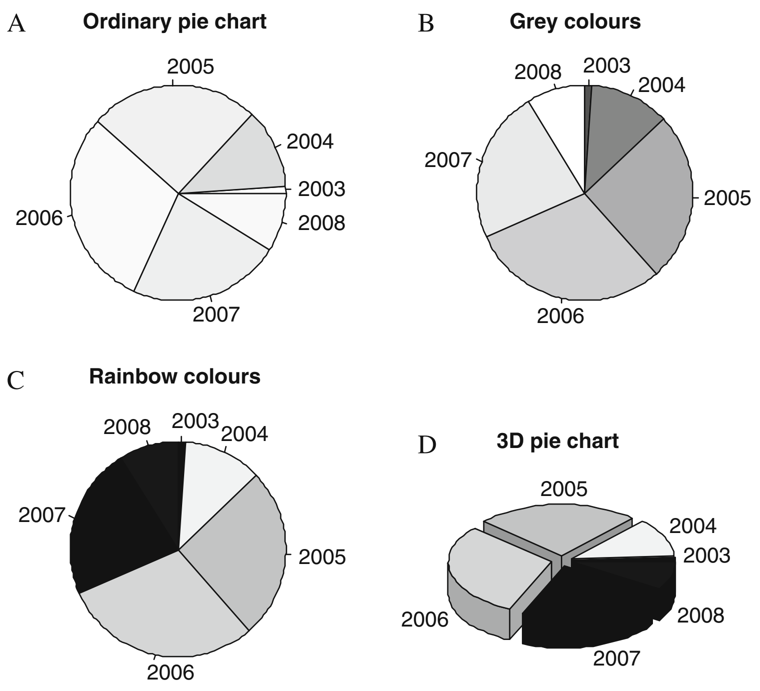


Fig. 7.1 **A**: Standard pie chart. **B**: Pie chart with clockwise direction of the slices. **C**: Pie chart with rainbow colours (which have been converted to greyscale during the printing process). **D**: Three-dimensional pie chart

Figure 7.1 was made with the following R code.

```
> par(mfrow = c(2, 2), mar = c(3, 3, 2, 1))
> pie(Cases, main = "Ordinary pie chart")            #A
> pie(Cases, col = gray(seq(0.4, 1.0, length = 6)),
       clockwise = TRUE, main = "Grey colours")       #B
> pie(Cases, col = rainbow(6), clockwise = TRUE,
       main = "Rainbow colours")                      #C
> library(plotrix)
> pie3D(Cases, labels = names(Cases), explode = 0.1,
         main = "3D pie chart", labelcex = 0.6)       #D
```

The `par` function is discussed in the next section. The variable `Cases` is of length 6 and contains totals per year. The command `pie (Cases)` creates the pie chart in Fig. 7.1A. Note that the direction of the slices is anticlockwise, which may be awkward, because our variable is time related. We reversed this in the second pie chart (Fig. 7.1B) with the option `clockwise = TRUE`. We also changed the colours, but, because this book is printed without colour, try this yourself: type in the code and see the colours of the pie charts in panels A–C. Because most of your work is likely to end up in a greyscale paper or report, we recommend using greyscale from the beginning. The only exception is for a PowerPoint presentation, where it is useful to present coloured pie charts. Note that the term “useful” refers to “coloured,” rather than to pie charts per se. The main problem with the pie chart is illustrated in Fig. 7.1: Although 2005 and 2006 have the largest slices, it is difficult to determine whether you should stay at home and close the windows and doors to survive the next pandemic, or whether “only” a handful of people were unfortunate enough to contract the disease. The pie chart does not give information on sample size.

Finally, Fig. 7.1D shows a three-dimensional pie chart. Although it now looks more like a real pie, it is, if anything, even less clear in its presentation than the other three graphs. To make this graph, you need to install the package `plotrix`. The function `pie3D` has many options, and we suggest that you consult its help file to improve the readability of labels.

7.1.2 The `par` Function

The `par` function has an extensive list of graph parameters (see `?par`) that can be changed. Some options are helpful; others you may never use.

The `mfrow =c (2, 2)` creates a graphic window with four panels. Changing the `c (2, 2)` to `c (1, 4)` or `c (4, 1)` produces a row (or column) of four pie charts. If you have more than four graphs, for instance 12, use `mfrow =c(3, 4)`, although now things can become crowded.

The `mar` option specifies the amount of white space around each graph (each pie chart in this case). The white space is defined by the number of lines of margin at the four sides; bottom, left, top, and right. The default values are, respectively, `c (5, 4, 4, 2) + 0.1`. Increasing the values gives more white space. Using trial and error, we chose `c (3, 3, 2, 1)`.

A problem arises with the `par` function if you execute the code for the four pie charts above and, subsequently, make another graph. R is still in the 2 × 2 mode, and will overwrite Figure 7.1A, leaving the other three graphs as they are. The next graph will overwrite panel B, and so on. There are two ways to avoid this. The first option is simply to close the four-panel graph in R before making a new one. This is a single mouse click. The alternative is a bit more programming intensive:

```
> op <- par(mfrow = c(2, 2), mar = c(3, 3, 2, 1))
> pie(Cases, main = "Ordinary pie chart")
> pie(Cases, col = gray(seq(0.4, 1.0, length = 6)),
      clockwise = TRUE, main = "Grey colours")
> pie(Cases, col = rainbow(6), clockwise = TRUE,
      main = "Rainbow colours")
```

```
> pie3D(Cases, labels = names(Cases), explode = 0.1,
      main = "3D pie chart", labelcex = 0.6)
> par(op)
```

The graph parameter settings are stored in the variable `op` on the first line. The graphs are made as before, and the last line of code converts to the default settings. Any new graph created after the `par` (`op`) command will be plotted as if the `par` function had not been used. This is useful if you need to create many graphs in sequence. It is neat programming, but takes more typing. It is often tempting to be lazy and go for the first approach. However, for good programming practice, we recommend making the extra effort. You will also see this style of programming in the help files.

Do Exercise 1 in Section 7.10 using the `pie` function.

7.2 The Bar Chart and Strip Chart

We give two examples of the bar chart, another type of graph that is not part of our toolbox. In the first example, we continue with the avian influenza data and present a bar chart showing the total number of bird flu cases and deaths per year. In the second example, a marine benthic dataset is used, with mean values per beach plotted as bars. In the last section, we show a strip chart to visualise similar information.

7.2.1 The Bar Chart Using the Avian Influenza Data

In the previous section, an avian influenza dataset was used to create pie charts showing the total number of cases per year. In addition to bird flu cases, the number of deaths is also available and can be found in the tab-separated ascii file, *Birdfludeaths.txt*. The data are loaded with the commands:

```
> BFDeaths <- read.table(file = "Birdfludeaths.txt",
                         header = TRUE)
> Deaths <- rowSums(BFDeaths[, 2:16])
> names(Deaths) <- BFDeaths[, 1]
> Deaths

2003 2004 2005 2006 2007 2008
   4   32   43   79   59   26
```

The data are structured in the same manner as the bird flu cases. We can visualise the change in the number of cases over time, and then compare number of cases to deaths.

The bar chart in Fig. 7.2A shows the change in the number of cases over time using the data from the variable `Cases` (see Section 7.1 for code to calculate `Cases`). Recall that `Cases` has six values with the labels 2003–2008. Each year is presented as a vertical bar. This graph is more useful than the pie chart, as we can read the absolute values from the *y*-axis. However, a great deal of ink and space is consumed by only six values.

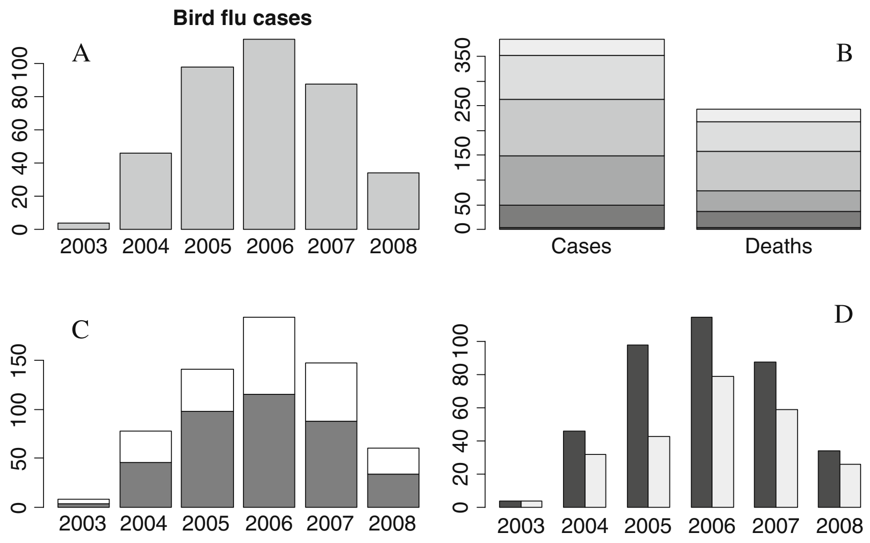


Fig. 7.2 **A**: Standard bar chart showing the annual number of bird flu cases. **B**: Stacked bar chart showing the accumulated totals per year for cases and deaths (note that values for 2003 can hardly be seen). **C**: Stacked cases (*grey*) and deaths (*white*) per year. **D**: Number of cases and deaths per year represented by adjoining bars

The first two lines of the code below were used to make the bar chart in panel A. The remaining code is for panels B–D:

```
> par(mfrow = c(2, 2), mar = c(3, 3, 2, 1))
> barplot(Cases , main = "Bird flu cases")             #A
> Counts <- cbind(Cases, Deaths)
> barplot(Counts)                                      #B
> barplot(t(Counts), col = gray(c(0.5, 1)))            #C
> barplot(t(Counts), beside = TRUE)                    #D
```

In panels B–D, we used the combined data for cases and deaths; these are called `Counts` and are of dimension 6 × 2:

```
> Counts
       Cases  Deaths
2003     4       4
2004    46      32
2005    98      43
2006   115      79
2007    88      59
2008    34      26
```

In panel B the bars represent data for each year. The graph gives little usable information. Also, years with small numbers (e.g., 2003) are barely visible. To produce panel C, we took the transposed values of `Counts` using the function `t`, making the input for the `barplot` function a matrix of dimension 2 × 6.

```
> t(Counts)
        2003 2004 2005 2006 2007 2008
Cases      4   46   98  115   88   34
Deaths     4   32   43   79   59   26
```

Although you see many such graphs in the literature, they can be misleading. If you compare the white boxes with one another, your eyes tend to compare the values along the *y*-axis, but these are affected by the length of the grey boxes. If your aim is to show that in each year there are more cases than deaths, this graph may be sufficient (comparing compositions). Among the bar charts, panel D is probably the best. It compares cases and deaths within each year, and, because there are only two classes per year, it is also possible to compare cases and deaths among years.

7.2.2 A Bar Chart Showing Mean Values with Standard Deviations

In Chapter 27 of Zuur et al. (2007), core samples were taken at 45 stations on nine beaches along the Dutch coastline. The marine benthic species were determined in each sample with over 75 identified. In Chapter 6, we developed a function to calculate species richness, the number of different species. The file *RIKZ2.txt* contains the richness values for the 45 stations and also a column identifying the beach.

The following R code imports the data and calculates the mean richness and standard deviation per beach. The `tapply` function was discussed in Chapter 4[1].

[1] Note that we could have omitted the text `INDEX =` and `FUN =`.

```
> setwd("C:/RBook/")
> Benthic <- read.table(file = "RIKZ2.txt",
                        header = TRUE)
> Bent.M <- tapply(Benthic$Richness,
                   INDEX = Benthic$Beach, FUN = mean)
> Bent.sd <- tapply(Benthic$Richness,
                   INDEX = Benthic$Beach, FUN = sd)
> MSD <- cbind(Bent.M, Bent.sd)
```

The variable `Bent.M` contains the mean richness values, and `Bent.sd` the standard deviation, for each of the nine beaches. We combined them in a matrix `MSD` with the `cbind` command. The values are as follows:

```
> MSD

 Bent.M Bent.sd
1   11.0 1.224745
2   12.2 5.357238
3    3.4 1.816590
4    2.4 1.341641
5    7.4 8.532292
6    4.0 1.870829
7    2.2 1.303840
8    4.0 2.645751
9    4.6 4.393177
```

To make a graph in which the mean values are plotted as a bar and the standard deviations as vertical lines extending above the bars (Fig. 7.3A) use the following procedure. For the graph showing mean values, enter

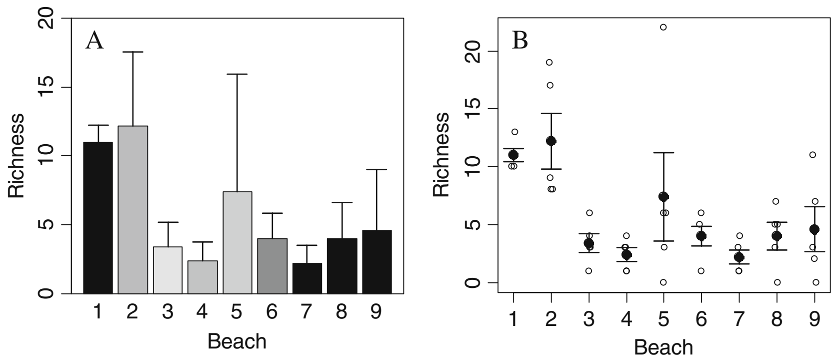


Fig. 7.3 **A**: Bar chart showing the benthic data. Mean values are represented by the *bars* with a *vertical line* showing standard deviations. The colours were changed to greyscale during the printing process. **B**: Strip chart for the raw data. The mean value per beach is plotted as a *filled dot*, and the *lines* represent the mean +/– the standard error

```
> barplot(Bent.M)
```

Add labels and perhaps some colour for interest:

```
> barplot(Bent.M, xlab = "Beach", ylim = c(0, 20),
          ylab = "Richness", col = rainbow(9))
```

The vertical lines indicating standard deviations are added using the function `arrows` to draw an arrow between two points with coordinates (x_1, y_1) and (x_2, y_2). Telling R to draw an arrow between the points (x, y_1) and (x, y_2), will produce a vertical arrow, as both points have the same x-value. The y_1-value is the mean, and the y_2-value is the mean plus the standard deviation. The x is the coordinate of the midpoint of a bar. The following code obtains these values and creates Fig. 7.3A.

```
> bp <- barplot(Bent.M, xlab = "Beach", ylim = c(0,20),
                ylab = "Richness", col = rainbow(9))
> arrows(bp, Bent.M, bp, Bent.M + Bent.sd, lwd = 1.5,
         angle = 90, length = 0.1)
> box()
```

It is the `bp <–barplot (Bent.M, ...)` that helps us out. The best way to understand what it does is by typing:

```
> bp
```

```
      [,1]
 [1,] 0.7
 [2,] 1.9
 [3,] 3.1
 [4,] 4.3
 [5,] 5.5
 [6,] 6.7
 [7,] 7.9
 [8,] 9.1
 [9,] 10.3
```

They are the midpoints along the x-axis of each bar, which are used as input for the `arrows` function. The `angle = 90` and `length = 0. 1` options change the head of the arrow into a perpendicular line. The `lwd` stands for line width with 1 as the default value. The `box` function draws a box around the graph. Run the code without it and see what happens.

7.2.3 The Strip Chart for the Benthic Data

In the previous section, a marine benthic dataset was used, and the mean species richness values per beach were presented as bars with a line representing standard deviation. Section 7.4 in Zar (1999) contains a discussion of when to present the standard deviation, standard error, or twice the standard error (assuming a large sample). It is relatively easy to produce a graph with the raw data, mean values, and either the standard deviation or standard error around the mean. An example is given in Fig. 7.3B. Instead of using the `plot` function, we used the `stripchart` function. The open dots show the raw data. We have added random jittering (variation) to distinguish observations with the same value, which would otherwise coincide. The filled dots are the mean values per beach, and were calculated in the previous section. We illustrate the standard errors, which are calculated by dividing the standard deviation by the square root of the sample size (we have five observations per beach). In R, this is done as follows.

```
> Benth.le <- tapply(Benthic$Richness,
                INDEX = Benthic$Beach, FUN = length)
> Bent.se <- Bent.sd / sqrt(Benth.le)
```

The variable `Bent.se` now contains the standard errors. Adding the lines for standard error to the graph is now a matter of using the `arrow` function; an arrow is drawn from the mean to the mean plus the standard error, and also from the mean to the mean minus the standard error. The code is below.

```
> stripchart(Benthic$Richness ~ Benthic$Beach,
       vert = TRUE, pch = 1, method = "jitter",
       jit = 0.05, xlab = "Beach", ylab = "Richness")
> points(1:9, Bent.M, pch = 16, cex = 1.5)
> arrows(1:9, Bent.M,
         1:9, Bent.M + Bent.se, lwd = 1.5,
         angle = 90, length = 0.1)
> arrows(1:9, Bent.M,
         1:9, Bent.M - Bent.se, lwd = 1.5,
         angle = 90, length = 0.1)
```

The options in the `stripchart` function are self-explanatory. Change them to see what happens. The `points` function adds the dots for the mean values. Instead of the `stripchart` function, you can also use the `plot` function, but it does not have a `method = "jitter"` option. Instead you can use `jitter (Benthic$Richness)`. Similar R code is given in Section 6.1.3 in Dalgaard (2002).

Do Exercise 2 in Section 7.10. This is an exercise in the `barchart` and `stripchart` functions using vegetation data.

7.3 Boxplot

7.3.1 Boxplots Showing the Owl Data

The boxplot should most often be your tool of choice, especially when working with a continuous numerical response (dependent) variable and categorical explanatory (independent) variables. Its purpose is threefold: detection of outliers, and displaying heterogeneity of distribution and effects of explanatory variables. Proper use of this graphing tool, along with the Cleveland dotplot (which is described fully in Section 7.4), can provide a head start on analysis of data.

In Chapter 6 we used a dataset on owl research. Roulin and Bersier (2007) looked at how nestlings respond to the presence of the father and of the mother. Using microphones inside and a video outside the nests, they sampled 27 nests, and studied vocal begging behaviour when the parents brought prey. Sampling took place between 21.30 hours and 05.30 hours on two nights. Half the nests were food deprived and the other half food satiated (this was reversed on the second night). The variable `ArrivalTime` shows the time when a parent arrived at the perch with prey. "Nestling negotiation" indicates the average number of calls per nest.

One of the main questions posed is whether there is a feeding protocol effect and a sex of parent effect. The analysis requires mixed effects modelling techniques and is fully described in Zuur et al. (2009). Before doing any complicated statistics, it is helpful to create boxplots. A boxplot for the nestling negotiation data is easily made using the `boxplot` function seen in Chapter 1. In Chapter 6, we showed the output of the `names` and `str` functions for the owl data, and do not repeat it here.

```
> setwd("C:/RBook/")
> Owls <- read.table(file = "Owls.txt", header = TRUE)
> boxplot(Owls$NegPerChick)
```

The resulting graph is presented in Fig. 7.4. A short description of the boxplot construction is given in the figure labelling. There are five potential outliers, indicating that further investigation is required.

Figure 7.5 illustrates possible effects of sex of the parent (panel A), food treatment (panel B), and the interaction between sex of the parent and food treatment (panels C and D). Because the variable names are long, they are not completely displayed in panel C. We reproduced the boxplots from panel C in panel D and added labels using the `names` option. Results indicate that there is a possible food treatment effect. The interaction is not clear, which was confirmed by formal statistical analysis. The R code to make Fig. 7.4 is given below. Panels C and D were produced with the `SexParent *FoodTreatment` construction. The code is self-explanatory.

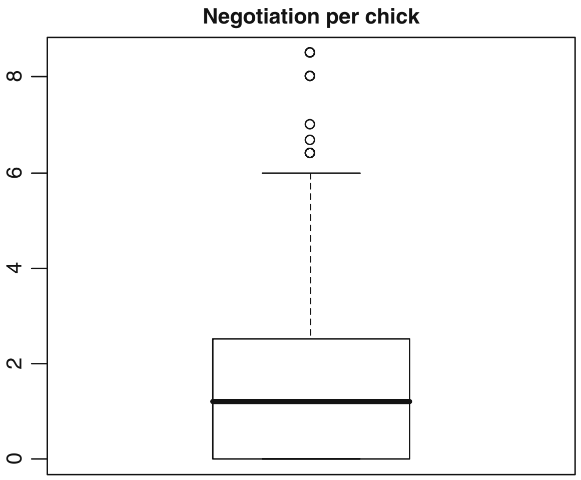


Fig. 7.4 Boxplot of owl nestling negotiation. The *thick horizontal line* is the median; the box is defined by the 25th and 75th percentiles (*lower* and *upper* quartile). The difference between the two is called the spread. The *dotted line* has a length of 1.5 times the spread. (The length of the line pointing up is shorter if the values of the points are smaller than the 75th percentile + 1.5 × spread, and similar for the line pointing downwards. This explains why there is no line at the bottom of the box.) All points outside this range are potential outliers. See Chapter 4 in Zuur et al. (2007) for a discussion of determining if such points are indeed outliers. Note that in this case the 25th percentile is also the smallest value (there are many zero values)

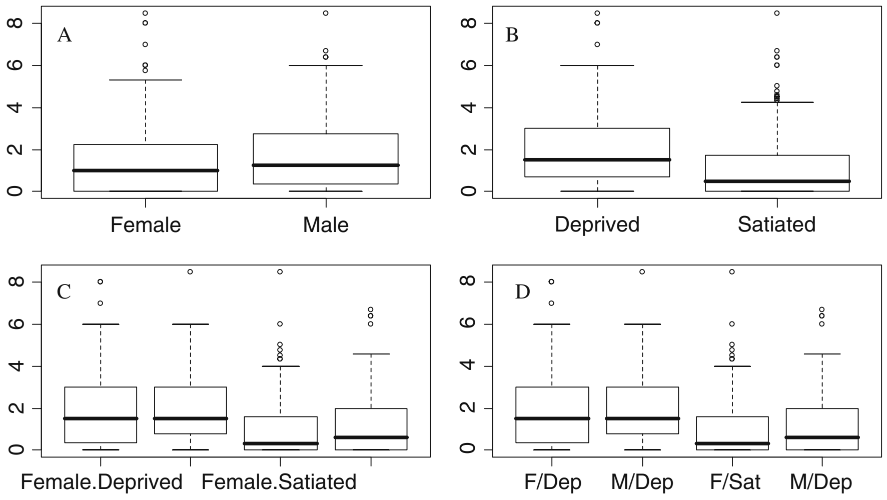


Fig. 7.5 **A**: Boxplot of owl nestling negotiation conditional on sex of the parent. **B**: Boxplot of owl nestling negotiation conditional on food treatment. **C**: Boxplot of owl nestling negotiation conditional on sex of the parent and food treatment. **D**: Same as panel C, with added labels

```
> par(mfrow = c(2,2), mar = c(3, 3, 2, 1))
> boxplot(NegPerChick ~ SexParent, data = Owls)
> boxplot(NegPerChick ~ FoodTreatment, data = Owls)
> boxplot(NegPerChick ~ SexParent * FoodTreatment,
          data = Owls)
> boxplot(NegPerChick ~ SexParent * FoodTreatment,
        names = c("F/Dep", "M/Dep", "F/Sat", "M/Sat"),
        data = Owls)
```

Sometimes getting all the labels onto a boxplot calls for more creativity. For example, Fig. 7.6 shows a boxplot of nestling negotiation conditional on nest. There are 27 nests, all with long names. If we had entered

```
> boxplot(NegPerChick ~ Nest, data = Owls)
```

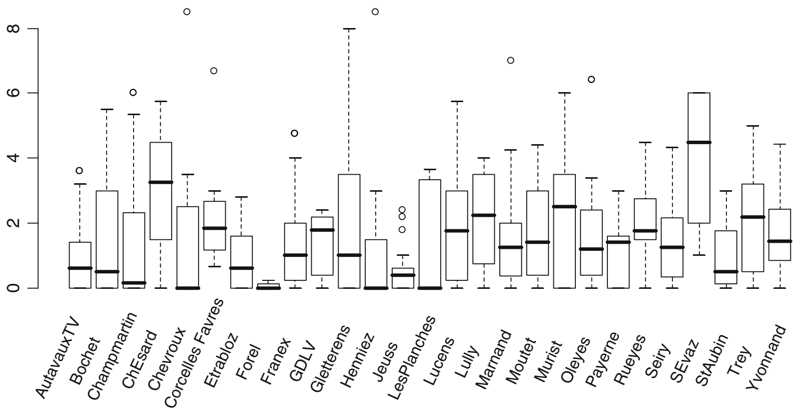


Fig. 7.6 Boxplot of owl nestling negotiation conditional on the 27 nests. The shape of the boxplot suggests that there may be a nest effect, suggesting further analysis by mixed effects models

only a few of the labels would be shown. The solution was to create the boxplot without a horizontal axis line and to put the labels in a small font, at an angle, under the appropriate boxplot. This sounds complicated, but requires only three lines of R code.

```
> par(mar = c(2, 2, 3, 3))
> boxplot(NegPerChick ~ Nest, data = Owls,
          axes = FALSE, ylim = c (-3, 8.5))
> axis(2, at = c(0, 2, 4, 6, 8))
> text(x = 1:27, y = -2, labels = levels(Owls$Nest),
        cex = 0.75, srt = 65)
```

Because we used the option `axes =FALSE`, the `boxplot` function drew the boxplot without axes lines. The `ylim` specifies the lower and upper limits of the vertical axis. Instead of using limits from 0 to 8.5, we used –3 to 8.5. This allowed us to put the labels in the lower part of the graph (Fig. 7.6).

The `axis` function draws an axis. Because we entered 2 as the first argument, the vertical axis on the left is drawn, and the `at` argument specifies where the tick marks should be. The `text` command places all the labels at the appropriate coordinates. The `cex` argument specifies the font size (1 is default) and `srt` defines the angle. You will need to experiment with these values and choose the most appropriate settings.

7.3.2 Boxplots Showing the Benthic Data

Recall that in the marine benthic dataset, species richness was measured at nine beaches. We now make a boxplot for each beach (Fig. 7.7). Note that there are only five observations per beach. Because this is a low number for boxplots, we want to add information on sample size per beach to the graph. One option is to specify the `varwidth = TRUE` option in the `boxplot` function to make the width of each box proportional to the number of observations on the beach. However, we instead choose to add the number of samples per beach inside each box. First, we need to obtain the sample size per beach using the following R code.

```
> setwd("C:/RBook/")
> Benthic <- read.table(file = "RIKZ2.txt",
                        header= TRUE)
> Bentic.n <- tapply(Benthic$Richness, Benthic$Beach,
                     FUN = length)
> Bentic.n
```

```
1 2 3 4 5 6 7 8 9
5 5 5 5 5 5 5 5 5
```

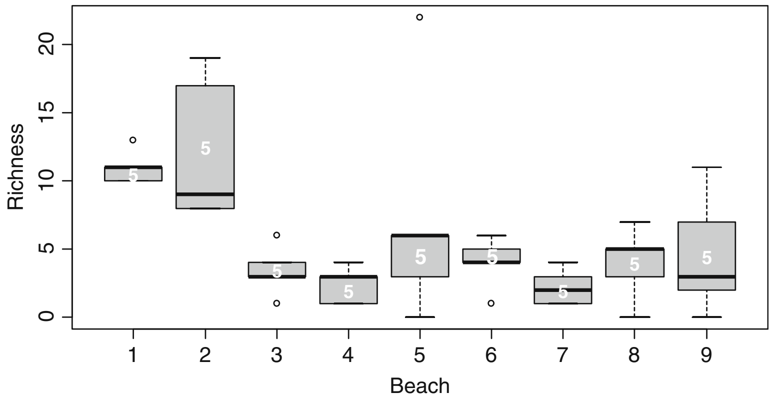


Fig. 7.7 Conditional boxplot using species richness as the dependent variable and beach as the conditioning variable. Number of observations per beach is shown inside each box

The `tapply` function calculates the number of observations per beach, 5, and stores them in the variable `Benthic.n`. The boxplot is created with the command

```
> boxplot(Richness ~ Beach, data = Benthic,
      col = "grey", xlab = "Beach", ylab = "Richness")
```

There is no new code here. The problem is placing the numbers of the variable `Benthic.n` inside the boxplot, preferably in the centre (which is not necessarily the median). Recall that the box is specified by the upper and lower quartiles. Adding half the value of the spread (upper hinge minus lower hinge) to the value of the lower hinge will put us vertically centred in the boxplot. Fortunately, all these values are calculated by the `boxplot` function and can be stored in a list by using

```
> BP.info <- boxplot(Richness ~ Beach, data = Benthic,
                   col = "grey", xlab = "Beach",
                   ylab = "Richness")
```

The list `BP.info` contains several variables, among them `BP.info $stats`. The `boxplot` help file will tell you that the second row of `$stats` contains the values of the lower hinges (for all beaches), and the fourth row shows the upper hinges. Hence, the midpoints (along the vertical axes) for all beaches are given by:

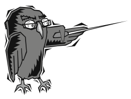
```
> BP.midp <- BP.info$stats[2, ] +
        (BP.info$stats[4, ] - BP.info$stats[2,]) / 2
```

It is now easy to place the numbers in `Bentic.n` inside the boxplot:

```
> text(1:9, BP.midp, Bentic.n, col = "white", font = 2)
```

We can put any text into the boxplot with this construction. For longer strings, you may want to rotate the text 90 degrees.

The `boxplot` function is very flexible and has a large number of attributes that can be changed. Have a look at the examples in the help files of `boxplot` and `bxp`.



Do Exercises 3 and 4 in Section 7.10. These are exercises in the `boxplot` function using the vegetation data and a parasite dataset.

7.4 Cleveland Dotplots

Dotplots, also known as Cleveland dotplots, are excellent tools for outlier detection. See Cleveland (1993), Jacoby (2006), or Zuur et al. (2007, 2009) for examples.

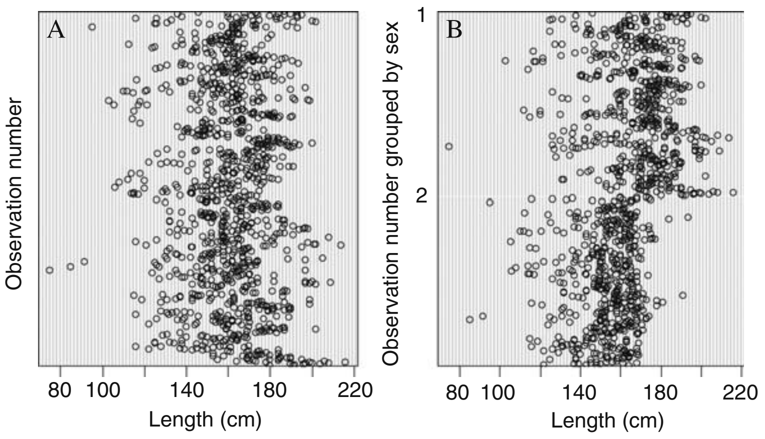


Fig. 7.8 **A**: Cleveland dotplot showing deer length. The x-axis shows the length value and the y-axis is the observation number (imported from the ascii file). The first observation is at the bottom of the y-axis. **B**: As panel **A**, but with observations grouped according to sex. There may be correlation between length and sex.

Figure 7.8 contains two dotplots for the deer dataset (Vicente et al., 2006), which was used in Section 4.4. Recall that the data were from multiple farms, months, years, and sexes. One of the aims of the study was to assess the relationship between the number of *E. cervi* parasites in deer and the length of the animal. Before doing any analysis, we should inspect each continuous variable in the dataset for outliers. This can be done with a boxplot or with a Cleveland dotplot. Figure 7.8A shows a Cleveland dotplot for the length of the animals. The majority of the animals are around 150 centimetres in length, but there are three animals that are considerably smaller (around 80 centimetres). As a consequence, applying a generalised additive model using length as a smoother may result in larger confidence bands at the lower end of the length gradient.

You can extend a Cleveland dotplot by grouping the observations based on a categorical variable. This was done in Fig. 7.8B; the length values are now grouped by sex. Note that one sex class is clearly larger. The goal of the study was to model the number of parasites (*E. cervi*) as a function of length, sex, year, and farm, in order to determine which of the explanatory (independent) variables is the crucial factor. However, it is difficult to say which explanatory variable is important if there are correlations among the variables. Such a situation is called collinearity. In this case, visualizing length versus sex is useful and can be done with a boxplot in which length is plotted conditional on sex, or with the Cleveland dotplot (Fig. 7.8B).

The graphs were created using the R function, `dotchart`. Function `dotchart2` in the package `Hmisc` (which is not part of the base installation) can produce more sophisticated presentations. We limit our discussion to `dotchart`. The data are imported with the following two lines of code.

```
> setwd("C:/RBook/")
> Deer <- read.table("Deer.txt", header = TRUE)
```

We have seen the output of the `names` and `str` commands in Section 4.4, and this information is not repeated. The Cleveland dotplot in Fig. 7.8A is produced with the following R code.

```
> dotchart(Deer$LCT, xlab = "Length (cm)",
           ylab = "Observation number")
```

The `dotchart` function has various options. The `groups` option allows grouping the data by categorical variable:

```
> dotchart(Deer$LCT, groups = factor(Deer$Sex))

Error in plot.window(xlim, ylim, log, asp, ...) :
       need finite 'ylim' values
```

The variable `Sex` has missing values (type `Deer $Sex` in the R console to view them), and, as a result, the `dotchart` function stops and produces an error message. The missing values can easily be removed with the following code.

```
> Isna <- is.na(Deer$Sex)
> dotchart(Deer$LCT[!Isna],
           groups = factor(Deer$Sex[!Isna]),
         xlab = "Length (cm)",
         ylab = "Observation number grouped by sex")
```

The `is.na` function produces a vector of the same length as `Sex`, with the values `TRUE` and `FALSE`. The ! symbol reverses them, and only the values for which `Sex` is not a missing value are plotted. Note that we used similar code in Chapter 3. If you want to have the two Cleveland dotplots in one graph, put the `par (mfrow = c (1, 2))` in front of the first `dotchart`.

7.4.1 Adding the Mean to a Cleveland Dotplot

Cleveland dotplots are a good alternative to boxplots when working with small sample sizes. Figure 7.9A shows a Cleveland dotplot of the benthic data used earlier in this chapter. Recall that there are five observations per beach. The right graph shows the same information with the mean value for each beach added. This graph clearly shows at least one "suspicious" observation. The code is basic; see below. The first three commands import the data, with `Beach` defined as a factor. A graph window with two panels is created with the `par` function. The first `dotchart` command follows that of the deer data. To the second `dotchart` command, we have added the `gdata` and `gpch` options.

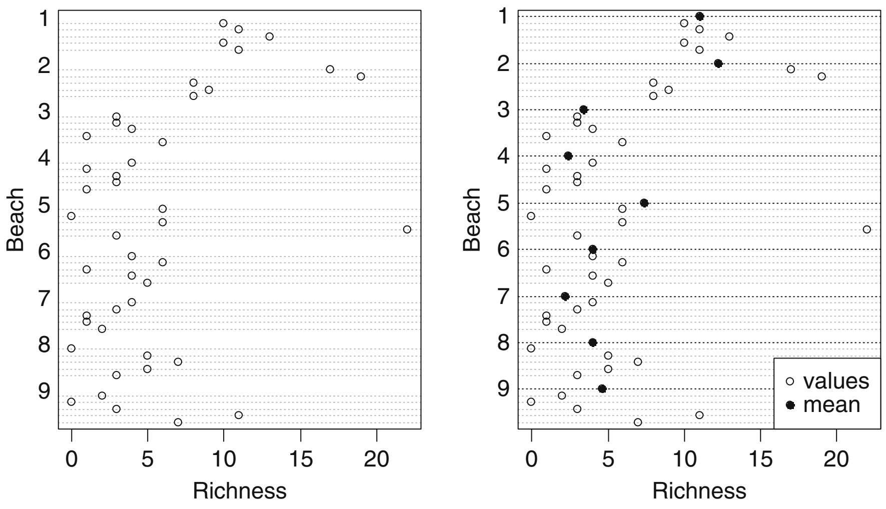
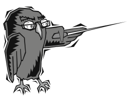


Fig. 7.9 Cleveland dotplots for the benthic data. **A**: The *vertical axis* shows the sampling sites, grouped by beach, and the *horizontal axis* the richness values. **B**: Same as A, with mean values per beach added

The g stands for group, and the `gdata` attribute is used to overlay a summary statistic such as the median, or, as we do here with the `tapply` function, the mean. Finally, the `legend` function is used to add a legend. We discuss the use of the `legend` function in more detail later in this chapter.

```
> setwd("C:/RBook/")
> Benthic <- read.table(file = "RIKZ2.txt",
                          header = TRUE)
> Benthic$fBeach <- factor(Benthic$Beach)
> par(mfrow = c(1, 2))
> dotchart(Benthic$Richness, groups = Benthic$fBeach,
           xlab = "Richness", ylab = "Beach")
> Bent.M<-tapply(Benthic$Richness, Benthic$Beach,
           FUN = mean)
> dotchart(Benthic$Richness, groups = Benthic$fBeach,
         gdata = Bent.M, gpch = 19, xlab = "Richness",
         ylab = "Beach")
> legend("bottomright", c("values", "mean"),
         pch = c(1, 19), bg = "white")
```



Do Exercises 5 and 6 in Section 7.10 creating Cleveland dotplots for the owl data and for the parasite data.

7.5 Revisiting the `plot` Function

7.5.1 The Generic `plot` Function

The most frequently used plotting command is `plot`, which was introduced in Chapter 5. It is an intuitive function, recognising what you intend to plot. R is an object-oriented language: the `plot` function looks at the object with which it is presented, establishes the object's class, and recruits the appropriate plotting method for that object. To see the methods available for a function (i.e., `plot`), enter

```
> methods(plot)
```

```
 [1] plot.acf*            plot.data.frame*  plot.Date*
 [4] plot.decomposed.ts* plot.default      plot.dendrogram*
 [7] plot.density        plot.ecdf         plot.factor*
[10] plot.formula*       plot.hclust*      plot.histogram*
[13] plot.HoltWinters*   plot.isoreg*      plot.lm
[16] plot.medpolish*     plot.mlm          plot.POSIXct*
[19] plot.POSIXlt*       plot.ppr*         plot.prcomp*
[22] plot.princomp*      plot.profile.nls* plot.spec
[25] plot.spec.coherency plot.spec.phase   plot.stepfun
[28] plot.stl*           plot.table*       plot.ts
[31] plot.tskernel*      plot.TukeyHSD
   Non-visible functions are asterisked
```

These are the existing plotting functions, and are only those available in the default packages. All these functions can be called with the `plot` function. For example, if you do a principal component analysis (PCA) and want to print the results, it is not necessary to use the `plot.princomp`, as the `plot` function will recognise that you conducted a PCA, and will call the appropriate plotting function. Another example is the following code.

```
> setwd("C:/RBook/")
> Benthic <- read.table(file = "RIKZ2.txt",
                        header = TRUE)
> Benthic$fBeach <- factor(Benthic$Beach)
> plot(Benthic$Richness ~ Benthic$fBeach)
```

The first three lines import the benthic dataset used earlier in this chapter and define the variable `Beach` as a factor. The `plot` function sees the formula `Benthic$Richness ~ Benthic$fBeach`, and produces a boxplot rather than a scatterplot (see the help file of `plot.factor`). If the argument in the `plot` function is a data frame, it will produce a pair plot (see Section 7.6).

7.5.2 More Options for the plot Function

In Chapter 5, we discussed the use of the `plot` function to plot two continuous variables against each other and also showed how to change the characters and colours. But there are many additional options, some of which we present in the remaining part of this section. We use the benthic data to demonstrate once again producing a scatterplot of two continuous variables (Fig. 7.10A). The graph was obtained with the following code.

```
> plot(y = Benthic$Richness, x = Benthic$NAP,
     xlab = "Mean high tide (m)",
     ylab = "Species richness", main = "Benthic data")
> M0 <- lm(Richness ~ NAP, data = Benthic)
> abline(M0)
```

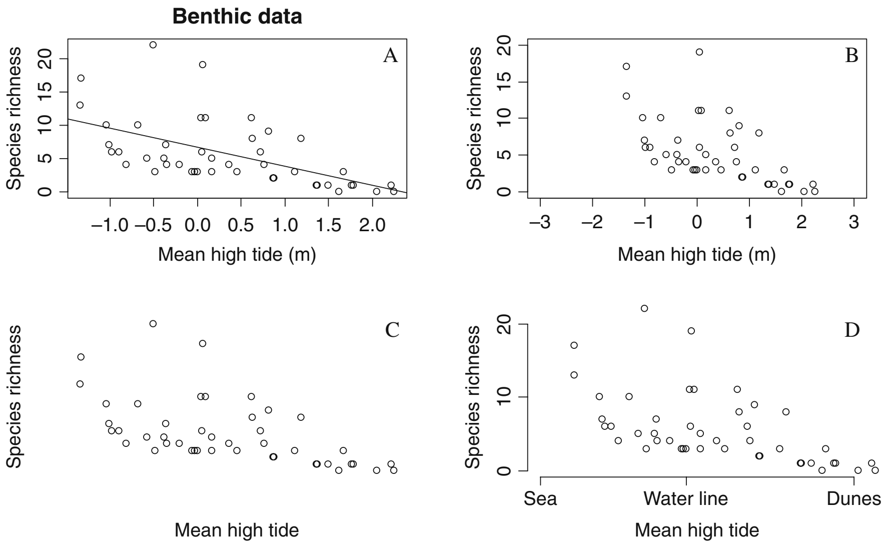


Fig. 7.10 **A**: Scatterplot of species richness versus NAP (mean high tide levels) with a linear regression line added. **B**: Same as panel **A**, with the *x*- and *y*-ranges set using the `xlim` and `ylim` functions. **C**: Same as panel A, but without axes lines. **D**: Same as panel **A**, with modified tick marks along the *y*-axis and character strings along the *x*-axis. Note that the sites are from an intertidal area, hence the negative values of mean high tide

The new addition is the `lm` and `abline` functions. Without going into statistical detail, the `lm` applies linear regression in which species richness is modelled as a function of NAP, the results are stored in the list `M0`, and the `abline` function superimposes the fitted line. Note that this only works if there is a single explanatory variable (otherwise, plotting the results in a two-dimensional graph becomes difficult), and if the `abline` function is executed following the `plot` function.

The `plot` function can easily be extended to add more detail to the graph by giving it extra arguments. Some of the most frequently used arguments are given in the table below.

Argument	What does it do?
`main`	Adds a title to the graph
`xlab, ylab`	Labels the x- and y- axis
`xlim, ylim`	Sets limits to the axes
`log`	Log = "x", log = "y", log = "xy" creates logarithmic axes
`type`	Type = "p", "l", "b", "o", "h", "s", "n" for plotting points, lines, points connected by lines, points overlaid by lines, vertical lines from points to the zero axis, steps, or only the axes

We have previously illustrated the `xlab` and `ylab` attributes. The `xlim` and `ylim` specify the ranges along the *x*- and *y*-axes. Suppose that you wish to set the range of the horizontal axis from –3 to 3 metres and the range along the vertical axis from 0 to 20 species. Use

```
> plot(y = Benthic$Richness, x = Benthic$NAP,
     xlab = "Mean high tide (m)",
     ylab = "Species richness",
     xlim = c(-3, 3), ylim = c(0,20))
```

The `xlim` argument has to be of the form `c(`x_1`, `x_2), with numerical values for x_1 and x_2. The same holds for the `ylim` argument. The results are shown in Fig. 7.10B.

Panels C and D in Fig. 7.10 show other options. Panel C does not contain axes lines. The R code is as follows.

```
> plot(y = Benthic$Richness, x = Benthic$NAP,
     type = "n", axes = FALSE,
     xlab = "Mean high tide",
     ylab = "Species richness")
> points(y = Benthic$Richness, x = Benthic$NAP)
```

The `type = n` produces a graph without points, and, because we use `axes = FALSE`, no axes lines are plotted. We begin with a blank window with only the labels. The `points` function superimposes the points onto the graph (note that you must execute the `plot` function prior to the `points` function or an error message will result).

In panel C, we basically told R to prepare a graph window, but not to plot anything. We can then proceed, in steps, to build the graph shown in Panel D. The `axis` function is the starting point in this process. It allows specifying the position, direction, and size of the tick marks as well as the text labelling them.

```
> plot(y = Benthic$Richness, x = Benthic$NAP,
    type = "n", axes = FALSE, xlab = "Mean high tide",
    ylab = "Species richness",
    xlim = c(-1.75,2), ylim = c(0,20))
> points(y = Benthic$Richness, x = Benthic$NAP)
> axis(2, at = c(0, 10, 20), tcl = 1)
> axis(1, at = c(-1.75, 0,2),
       labels = c("Sea", "Water line", "Dunes"))
```

The first two lines of code are identical to those for panel C. The `axis (2, ...)` command draws the vertical axis line and inserts tick marks of length 1 (the default value is –0.5) at the values 0, 10, and 20. Setting `tcl` to 0 eliminates tick marks. Tick marks pointing outwards are obtained by a negative `tcl` value; a positive value gives inward pointing tick marks. The `axis (1, ...)` command draws the horizontal axis, and, at the values –1.75, 0, and 2, adds the character strings Sea, Water line, and Dunes. See the `axis` help file for further graphing facilities.

Do Exercise 7 in Section 7.10. This is an exercise in the `plot` and `axis` functions using the owl data.

7.5.3 Adding Extra Points, Text, and Lines

This section addresses features that can be used to increase the visual appeal of graphs. Possible embellishments might be different types of lines and points, grids, legends, transformed axes, and much more. Look at the `par` help file, obtained by typing ?par, to see many of the features that can be added and altered. We could write an entire book on the `par` options, some of which have been addressed in Chapter 5 and in earlier sections of this chapter. More are discussed in Chapter 8. However, even novice users will feel the need for some information on the `par` function at an early point. Because we do not want this volume to become phonebook-sized, we discuss some of the `par` and plotting options in a birds-eye overview mode, and try to guide you to the appropriate help files.

The functions `points`, `text`, and `lines` are valuable companions when working in R and were used in some earlier chapters.

The function `points` adds new values to a plot, such as *x*-values and (optionally) *y*-values. By default, the function plots points, so, just as with `plot`, `type` is set to `"p"`. However, all the other types can be used: `"l"` for lines, `"o"` for overplotted points and lines, `"b"` for points and lines, `"s"` and `"S"` for steps, and `"h"` for vertical lines. Finally, `"n"` produces a graph-setup with no data points or lines (see Section 7.5.2). Symbols can be changed using `pch` (see Chapter 5).

The function text is similar to points in that it uses *x* and (optionally) *y*-coordinates but adds a vector called labels containing the label strings to be positioned on the graph. It includes extra tools for fine-tuning the placement of the string on the graph, for example, the attributes pos and offset. The pos attribute indicates the positions below, to the left of, above, and to the right of the specified coordinates (respectively, 1, 2, 3, 4) and offset gives the offset of the label from the specified coordinate in fractions of a character width. These two options become relevant with long character strings that are not displayed properly in R's default display.

We have seen the lines function in Chapter 5. It is a function that accepts coordinates and joins the corresponding points with lines.

7.5.4 *Using* `type = "n"`

With the `plot` function, it is possible to include the attribute `type = "n"` to draw everything but the data. The graph is set up for data, including axes and their labels. To exclude these, add `axes = FALSE, xlab = "", ylab = ""`. It then appears there is nothing left. However, this is not the case, because the plot retains the data that were entered in the first part of the `plot` function. The user is now in full control of constructing the plot. Do you want axes lines? If so, where do you want them and how do you want them to look? Do you want to display the data as points or as lines? Everything that is included in the default plot, and much more, can be altered and added to your plot. Here are some of the available variations:

Command	Description
`abline`	Adds an a,b (intercept, slope) line, mainly regression, but also vertical and horizontal lines
`arrows`	Adds arrows and modifies the head styles
`Axis`	Generic function to add an axis to a plot
`axis`	Adds axes lines
`box`	Adds different style boxes
`contour`	Creates a contour plot, or adds contour lines to an existing plot
`curve`	Draws a curve corresponding to the given function or expression
`grid`	Adds grid to a plot
`legend`	Adds legend to a plot
`lines`	Adds lines
`mtext`	Inserts text into the margins of the figure or in the margin of the plot device
`points`	Adds points, but may include `type` command
`polygon`	Draws polygons with vertices defined by x and y
`rect`	Draws rectangles
`rug`	Adds a one dimensional representation of the data to the plot on one of the two axes.
`Segments`	Adds line segments
`text`	Adds text inside the plot
`title`	Adds a title

7.5.5 Legends

The function legend appears difficult at first encounter, but is easily mastered. In Fig. 7.9, a legend was added to a Cleveland dotplot. The code is

```
> legend("bottomright", c("values", "mean"),
      pch = c(1, 19), bg ="white")
```

The first attribute may consist of an x- and y-coordinate, or an expression such as shown here. Other valid expressions are `"bottom"`, `"bottom-left"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"`, and `"center"`. Consult the `legend` help file for more options.

Zuur et al. (2009) used a bird dataset that was originally analysed in Loyn (1987), and again in Quinn and Keough (2002). Forest bird densities were measured in 56 forest patches in southeastern Victoria, Australia. The aim of the study was to relate bird densities to six habitat variables: (1) size of the forest patch, (2) distance to the nearest patch, (3) distance to the nearest larger patch, (4) mean altitude of the patch, (5) year of isolation by clearing, and (6) an index of stock grazing history (1 = light, 5 = intensive). A detailed analysis of these data using linear regression is presented in Appendix A of Zuur et al. (2009). The optimal linear regression model contained LOGAREA and GRAZE (categorical). To visualise what this model is doing, we plot the fitted values. There are five grazing levels, and, therefore, the linear regression (see the `summary` command below) gives an equation relating bird abundance to LOGAREA for each grazing level. These are given by

Observations with GRAZE = 1:	$ABUND_i = 15.7 + 7.2 \times LOGAREA_i$
Observations with GRAZE = 2:	$ABUND_i = 16.1 + 7.2 \times LOGAREA_i$
Observations with GRAZE = 3:	$ABUND_i = 15.5 + 7.2 \times LOGAREA_i$
Observations with GRAZE = 4:	$ABUND_i = 14.1 + 7.2 \times LOGAREA_i$
Observations with GRAZE = 5:	$ABUND_i = 3.8 + 7.2 \times LOGAREA_i$

Readers familiar with linear regression will recognise this as a linear regression model in which the intercept is corrected for the levels of the categorical variable. Next, we (i) plot the ABUNDANCE data versus LOGAREA, (ii) calculate fitted values for the five grazing regimes, (iii) add the five lines, and (iv) add a legend. The resulting graph is presented in Fig. 7.11. The following shows step by step how it was created.

First, read the data, apply the log transformation, and use the plot function. We have previously used similar R code:

```
> setwd("C:/RBook/")
> Birds <- read.table(file = "loyn.txt", header = TRUE)
> Birds$LOGAREA <- log10(Birds$AREA)
```

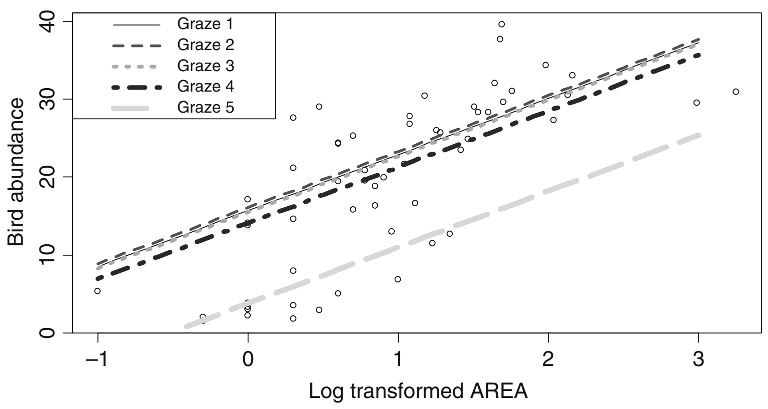


Fig. 7.11 Five fitted lines for the Loyn bird data. Each line is for a particular grazing regime

```
> plot(x = Birds$LOGAREA, y = Birds$ABUND,
     xlab = "Log transformed AREA",
     ylab = "Bird abundance")1
```

To see the source of the five slopes and the intercept, use the code:

```
> M0 <- lm(ABUND~ LOGAREA + fGRAZE, data = Birds)
> summary(M0)
```

If you are not familiar with linear regression, do not spend time struggling to comprehend this. The summary output contains the required information. To predict fitted bird abundances per grazing level, we need the LOGAREA values. The simplest method is to look at Fig. 7.11 and choose several arbitrary values within the range of the observed data, say –1, 0, 1, 2, and 3:

```
> LAR <- seq(from = -1, to = 3, by = 1)
> LAR

[1] -1 0 1 2 3
```

Now we determine the abundance values per grazing level using simple calculus and R code:

```
> ABUND1 <- 15.7 + 7.2 * LAR
> ABUND2 <- 16.1 + 7.2 * LAR
> ABUND3 <- 15.5 + 7.2 * LAR
> ABUND4 <- 14.1 + 7.2 * LAR
> ABUND5 <- 3.8 + 7.2  * LAR
```

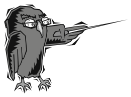
Adding the fitted values as lines to the graph is also familiar territory (see Chapter 5). We do not have a spaghetti problem, as the AREA data are sorted from – 1 to 3.

```
> lines(LAR, ABUND1, lty = 1, lwd = 1, col =1)
> lines(LAR, ABUND2, lty = 2, lwd = 2, col =2)
> lines(LAR, ABUND3, lty = 3, lwd = 3, col =3)
> lines(LAR, ABUND4, lty = 4, lwd = 4, col =4)
> lines(LAR, ABUND5, lty = 5, lwd = 5, col =5)
```

We added visual interest with different line types, widths, and colours. Finally, it is time to add the legend; see the R code below. First we define a string `legend.txt` with five values containing the text that we want to use in the legend. The `legend` function then places the legend in the top left position, the line in the legend for the first grazing level is black `(col = 1)`, solid `(lty = 1)`, and has normal line width `(lwd = 1)`. The line in the legend for grazing level 5 is light blue `(col = 5)`, has the form --- `(lty = 5)` and is thick `(lwd = 5)`.

```
> legend.txt <- c("Graze 1", "Graze 2",
                  "Graze 3", "Graze 4", "Graze 5")
> legend("topleft", legend = legend.txt,
       col = c(1, 2, 3, 4, 5),
       lty = c(1, 2, 3, 4, 5),
       lwd = c(1, 2, 3, 4, 5),
       bty = "o", cex = 0.8)
```

The attribute `cex` specifies the size of the text in the legend, and the `bty` adds a box around the legend.



Do Exercise 8 in Section 7.10. In this exercise, smoothers are used for the male and female owl data and are superimposed onto the graph. The `legend` function is used to identify them.

7.5.6 Identifying Points

The function identify is used to identify (and plot) points on a plot. It can be done by giving the *x*, *y* coordinates of the plot or by simply entering the plot object (which generally defines or includes coordinates). Here is an example:

```
> plot(y = Benthic$Richness, x = Benthic$NAP,
     xlab = "Mean high tide (m)",
     ylab = "Species richness", main = "Benthic data")
> identify(y = Benthic$Richness, x = Benthic$NAP)
```

With the attribute labels in the identify function, a character vector giving labels for the points can be included. To specify the position and offset of the labels relative to the points; place your mouse near a point and left-click; R will plot the label number close to the point. Press "escape" to cancel the process. It is also possible to use the identify function to obtain the sample numbers of certain points; see its help file. Note that the identify function only works for graphs created with the plot function, and not with boxplots, dotcharts, bar charts, pie charts, and others.

7.5.7 Changing Fonts and Font Size*

This section is a bit more specialised and may be skipped upon first reading. Fonts and font sizes are somewhat peculiar in R. When you open a graphing device you can apply an attribute pointsize that will be the default point size of plotted text. Default font mappings are provided for four device-independent font family names: "sans" for a sans-serif font, "serif" for a serif font, "mono" for a monospaced font, and "symbol" for a symbol font. Type windowsFonts() to see the font types that are currently installed.

Font defines the font face. It is an integer that specifies which font face to use for text. If possible, device drivers are organized so that 1 corresponds to plain text, 2 to bold face, 3 to italic, and 4 to bold italic. To modify the default font, we usually draw plots omitting the component for which we want to change the default font and code it separately, including options for font size, font face, and font family. For example, to add a title in a serif font to Fig. 7.11, use

```
> title("Bird abundance", cex.main = 2,
        family = "serif", font.main = 1)
```

This would plot "Bird abundance" as a title twice the default size, with a serif font style in normal font face. For title there are special options for font size and font face, cex.main and font.main. Sometimes you may need to specify the family using par. You can also change font and size for text, mtext, axis, xlab, and ylab. Consult the help file for par for specific information on changing fonts.

7.5.8 Adding Special Characters

Often you may want to include special characters in legends or labels. This is not difficult in R, although it may require searching in several help files to find exactly what you want. The function that is mostly used is expression. You can get an impression of the possibilities by typing demo(plotmath).

Here is a brief example: Mendes et al. (2007) measured the nitrogen isotopic composition in growth layers of teeth from 11 sperm whales stranded in Scotland. Figure 7.12 shows a scatterplot of nitrogen isotope ratios versus age, for one particular whale, nicknamed Moby. The *y*-label of the graph contains the expression δ^{15}N. It is tempting to import this graph without the *y*-label into Word and add the δ^{15}N before submission to a journal, but it can easily be done in R using this code:

```
> setwd("C:/RBook/")
> Whales <- read.table(file="TeethNitrogen.txt",
                       header = TRUE)
> N.Moby <- Whales$X15N[Whales$Tooth == "Moby"]
> Age.Moby <- Whales$Age[Whales$Tooth == "Moby"]
> plot(x = Age.Moby, y = N.Moby, xlab = "Age",
     ylab = expression(paste(delta^{15}, "N")))
```

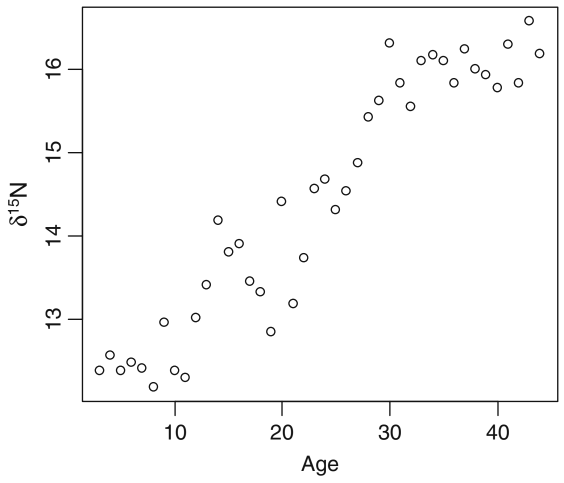


Fig. 7.12 Scatterplot of nitrogen isotope ratios versus age, as measured in dental growth layers of an individual whale, nicknamed Moby. Note the *y*-label

The `paste` command joins the δ^{15} and N, and the `expression` function inserts the δ^{15}N.

7.5.9 Other Useful Functions

There are a number of other functions that may come in handy when making graphs. Consult the help files for attributes that may, or must, be provided.

Function[a]	Description
`plot.new`	Opens a new graphics frame, same as frame()
`win.graph`	Opens extra second graph window. You can set width and height of the screen
`windows`	Similar to win.graph but with more options
`savePlot`	Saves current plot as ("wmf", "emf", "png", "jpeg", "jpg", "bmp", "ps", "eps", or "pdf")
`locator`	Records the position of the cursor by clicking left cursor; stops by clicking right cursor
`range`	Returns a vector containing the minimum and maximum of all the given arguments; useful for setting x or y limits
`matplot`	Plots columns of one matrix against the columns of another; especially useful when multiple Y columns and a single X. See also `matlines` and `matpoints` for adding lines and points, respectively
`persp`	Perspective plots of surfaces over an x–y plane
`cut`	Converts a numeric variable into a factor
`split`	Divides a vector or data frame with numeric values into groups

[a]Don't forget to include the brackets with these functions!

7.6 The Pairplot

In the previous graph, we used the `plot` function to make a scatterplot of two continuous variables; the following demonstrates scatterplots for multiple continuous variables. This could be done by using the `plot` function to plot variable 1 versus 2, 1 versus 3, 1 versus 4, and so on, and following with `mfrow` and `mar` to put it all into a single graph. However, the R function `pairs` can be used to produce a multipanel scatterplot. We use the benthic data for illustration:

```
> setwd("C:/RBook/")
> Benthic <- read.table(file = "RIKZ2.txt",
                        header = TRUE)
> pairs(Benthic[, 2:9])
```

The first two lines import the data, and the `pairs` function is applied to all the variables from the data frame `Benthic` with the exception of the first column, which contains the labels. The resulting graph is presented in Fig. 7.13[2].

We have included species richness as the first variable. As a result, the first row of the plot contains graphs of all variables against richness. The rest of the plot shows graphs of all variables versus one another. From a statistical point of view, we want to model richness as a function of all the other variables, hence

[2] Using the command `plot (Benthic [ , 2:9] )` will give the same graph, because `Benthic` is a data frame, and the `plot` function recognises this and calls the function `plot.data.frame`.

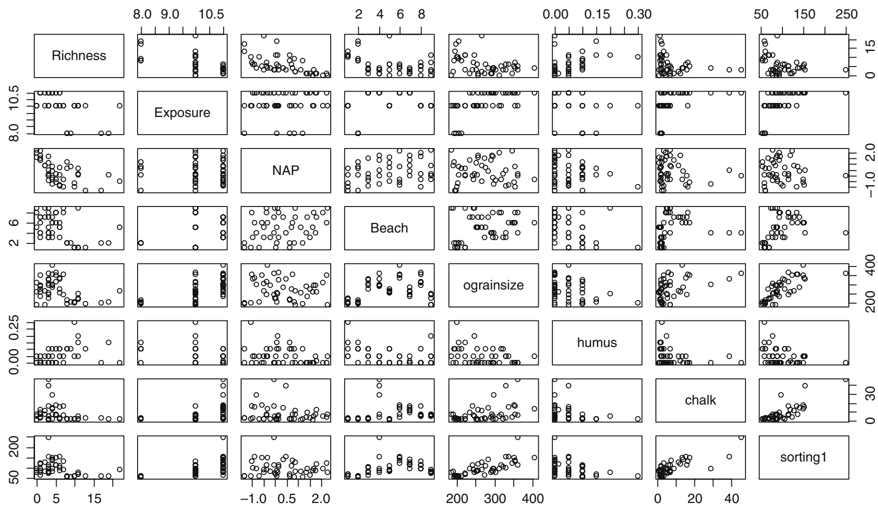


Fig. 7.13 Scatterplot matrix for variables in the benthic data. The diagonal shows the name of the variable which is on the *x*-axis below and above it, and on the *y*-axis left and right of it

clear relationships in the first row (or column) are good, whereas clear patterns in the other panels (collinearity) are not good at all. The pairplot shows clear relationships between some of the variables, for example, between species richness and NAP and between grain size and sorting (this makes biological sense, as sorting is a measure of energy).

7.6.1 Panel Functions

Half of the information in the pairplot appears superfluous, in as much as every graph appears twice, once above the diagonal and once below, but with the axes reversed. It is possible to specify panel functions to be applied to all panels, to the diagonal panels, or to the panels above or below the diagonal (Fig. 7.14). The R code for this can be found at the end of the `pairs` help file obtained by entering `?pairs` into the R console window.

```
> pairs(Benthic[, 2:9], diag.panel = panel.hist,
        upper.panel = panel.smooth,
        lower.panel = panel.cor)
Error  in  pairs.default(Benthic[,  2:9],  diag.panel  =
panel.hist, upper.panel = panel.smooth,: object
"panel.cor" not found
```

The problem here is that R does not recognise the `panel.cor` and the `panel.hist` functions. These specific pieces of code from the end of the

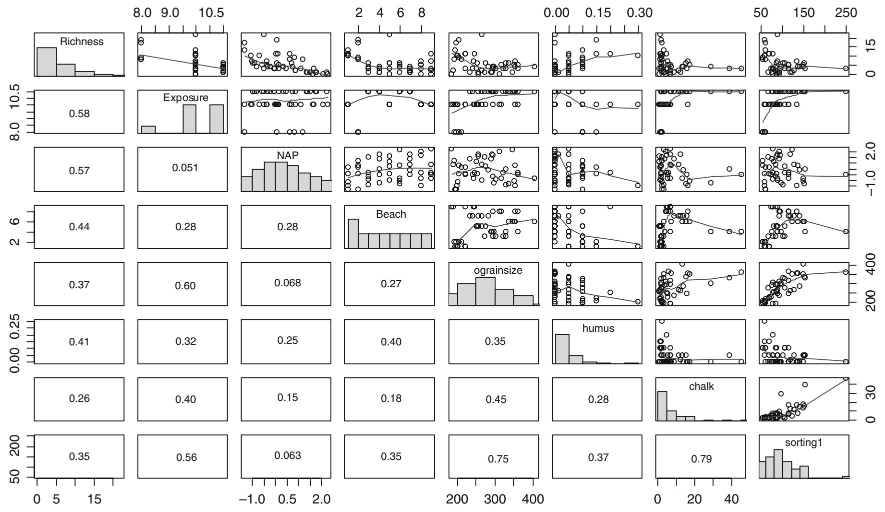
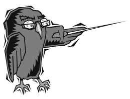


Fig. 7.14 The extended pairplot using histograms on the diagonal, scatter plots with smoothers above the diagonal, and Pearson correlation coefficients with size proportionate to the correlation below the diagonal. The code was taken from the `pairs` help file

`pairs` help file must be copied and pasted into the R console. Copy the entire function and rerun the `pairs` command above. For specific advice, see the online R code for this book, which can be found at www.highstat.com. The `panel.cor` and `panel.hist` code is complicated and beyond the scope of this book, so is not addressed here. Simply copy and paste it.

If you are interested in using Pearson correlation coefficients in a pairplot, see http://www.statmethods.net/graphs/scatterplot.html. This provides an example, as well as a link to the package and a function that can be used to colour entire blocks based on the value of the Pearson correlation.



Do Exercise 9 in Section 7.10. In this exercise, the `pairs` function is used for the vegetation data.

7.7 The Coplot

7.7.1 A Coplot with a Single Conditioning Variable

The `pairs` function shows only two-way relationships. The next plotting tools we discuss can illustrate three-way, or even four-way, relationships. This type of plot is called a conditioning plot or coplot and is especially well suited to visualizing how a response variable depends on a predictor, given other

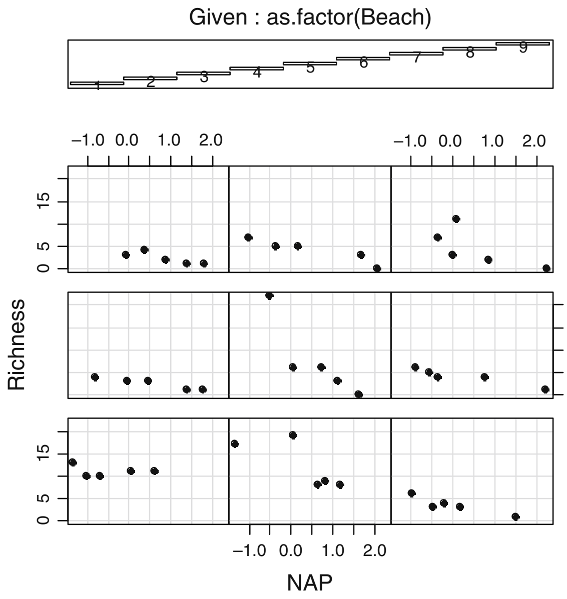


Fig. 7.15 Coplot for the benthic data. The *lower left* panel represents a scatter!plot of richness versus NAP for beach 1, the *lower right* panel for beach 3, the *middle left* for beach 4, and the *upper right* for beach 9

predictors. Figure 7.15 is a plot of the RIKZ data using the variables Beach, NAP, and Richness. The nine graphs represent beaches one to nine, which are listed at the top and displayed in the separate panels, called the dependence panels. Starting at the bottom row and going from left to right, the first row depicts beaches one to three, the second row four to six, and the top row beaches seven to nine. As you can see, the beach numbers are also given, although not well placed. The R code to make the graph in Fig. 7.15 is as follows.

```
> setwd("C:/RBook/")
> Benthic <- read.table(file = "RIKZ2.txt",
                              header = TRUE)
> coplot(Richness ~ NAP | as.factor(Beach), pch=19,
          data = Benthic)
```

The function `coplot` uses a different notation than the `plot` function. Variables to be plotted are given in a formula notation that uses the tilde operator ~ as a separator between the dependent and the independent variables. Contrary to what you have been using in the `plot` function where the first variable is assumed to be the x-variable and the second variable the y-variable, the formula notation always uses $y \sim x$. The above code thus directs R to plot species richness (R) versus NAP. The addition of | `as.factor (Beach)` creates the panels and indicates that the plot should be produced conditional on the variable Beach, which is first coerced into a factor. The `data` attribute gives the command to look in the `Benthic` data frame for the variables used in the formula.

Instead of using a categorical variable for the conditioning variable, we can use a continuous variable, for example, grainsize. The following code creates Fig. 7.16. Scatterplots of richness versus NAP are drawn for different grainsize values.

```
> coplot(Richness ~ NAP | grainsize, pch=19,
         data = Benthic)
```

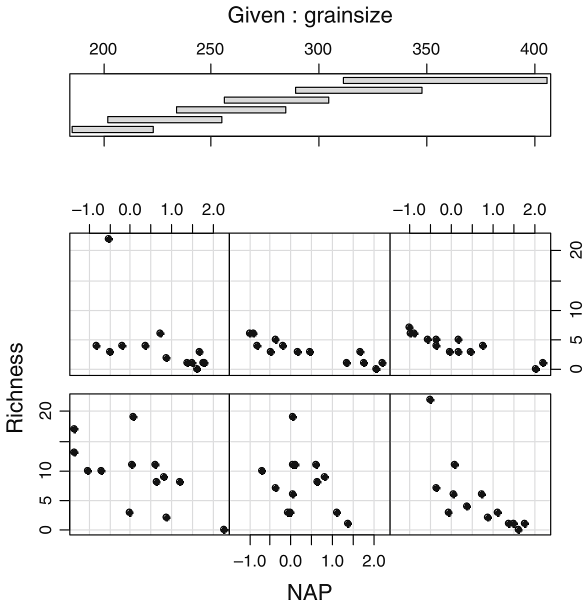


Fig. 7.16 Coplot using a continuous conditioning variable. The *lower left* panel represents a scatterplot of Richness versus NAP for those observations that have grainsize values between 185 and 220. The *upper right* panel shows a scatterplot of richness versus NAP for observations with high (>315) grainsize values. The important question is whether the Richness/NAP relationship changes along the grainsize gradient, indicating interaction between NAP and grainsize

The grainsize values were divided into six overlapping groups with approximately equal numbers of points. If the Richness/NAP relationship changes along the grainsize gradient, giving a visual indication of the presence of an interaction between NAP and grainsize, it may be worthwhile to include this interaction term in, for example, a linear regression model.

The `coplot` function contains a large number of arguments that can be used to create exciting plots. See its help file, obtained by `? coplot`. The most useful is `panel`, which takes a function that is carried out in each panel of the display. By default `coplot` uses the `points` function, but we can easily create our own function and apply it to each panel. For example, we may wish to add a linear regression line to each panel in Fig. 7.15 (Fig. 7.17). If all the lines turn out to be parallel, there is no visual evidence of an interaction between beach and NAP (i.e., the richness $\sim$ NAP relationship is the same along the entire stretch of coastline). In this case, the lines do differ. Here is the code that created Fig. 7.17:

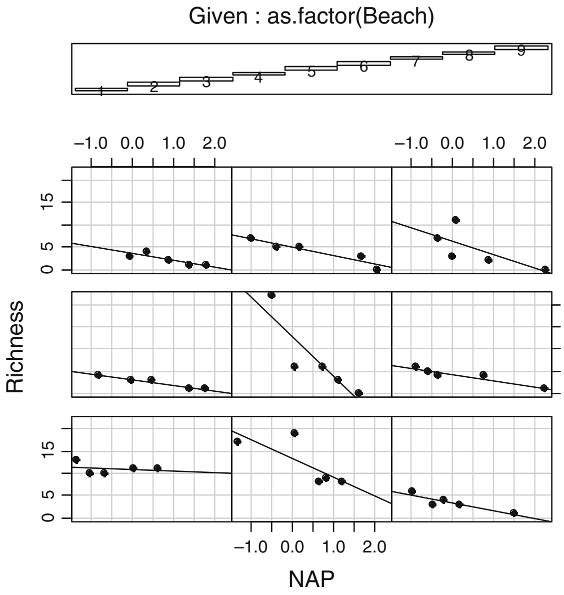


Fig. 7.17 Coplot of the RIKZ data showing species richness versus NAP with a separate panel for each of the nine beaches

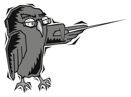
```
> panel.lm = function(x, y, ...) {
     tmp <- lm(y ~ x, na.action = na.omit)
     abline(tmp)
     points(x, y, ...)}
> coplot(Richness ~ NAP | as.factor(Beach), pch = 19,
         panel = panel.lm, data = Benthic)
```

The function `panel.lm` defines how the data should be displayed in each panel. Three dots at the end indicate that other arguments may be supplied that will be evaluated in the function. The linear regression function `lm` is used to store the data temporarily in the variable `tmp`, and any NAs are omitted from the analysis. The function `abline` plots the line, and the function `points` plots the points.

Another predefined `panel` function is `panel.smooth`. This uses the LOESS smoother to add a smooth line.

As you can see above, we defined our own `panel` function. This facility is useful for creating customized panel functions for use with `coplot`. For example, means and confidence limits can be added to each panel, and confidence limits can be added to regression lines.

`Coplot` is also a good tool for investigating the amount of data in each combination of covariates.



Do Exercise 10 in Section 7.10. This exercise creates a coplot of the vegetation data.

7.7.2 The Coplot with Two Conditioning Variables

One can include a third predictor variable in a coplot, but the benthic data do not yield much additional information when one of the other variables is included. Therefore we present another example: a subset of data analysed in Cruikshanks et al. (2006). The data are available in the file *SDI2003.txt*. The original research sampled 257 rivers in Ireland during 2002 and 2003. One of the aims was to develop a new tool for identifying acid-sensitive waters, which is currently done by measuring pH levels. The problem with pH is that it is extremely variable within a catchment and depends on both flow conditions and underlying geology. As an alternative measure, the Sodium Dominance Index (SDI) was proposed. Of the 257 sites, 192 were nonforested and 65 were forested. Zuur et al. (2009) modelled pH as a function of SDI, forested or nonforested, and altitude, using regression models with spatial correlation.

The relationship between pH and SDI may have been affected by the altitude gradient and forestation. Calculating this demands a three-way interaction term between two continuous (SDI and altitude) and one categorical (forestation) explanatory variable. Before including such an interaction in a model, we can visualise the relationships with the coplot. In the previous section, we used coplots with a single conditioning variable; here we use two conditioning variables. We use the log-transformed altitude values. The coplot is shown in Fig. 7.18. The R code is as follows.

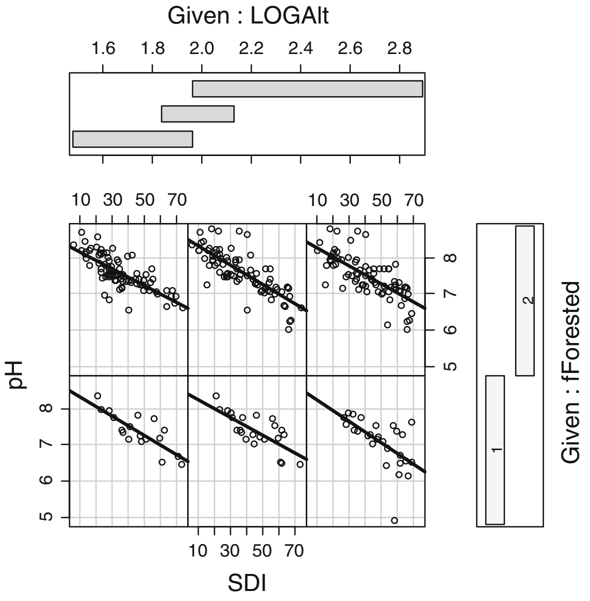


Fig. 7.18 Coplot of the Irish pH data. The panels show the relationship between pH and SDI for different altitudes and whether a site is forested. If the slopes of the lines vary, you will want to add an interaction term to the regression model. If a panel has no points, the interaction cannot be included

```
> setwd("C:/RBook/")
> pHEire <- read.table(file = "SDI2003.txt",
                       header = TRUE)
> pHEire$LOGAlt <- log10(pHEire$Altitude)
> pHEire$fForested <- factor(pHEire$Forested)
> coplot(pH ~ SDI | LOGAlt * fForested,
      panel = panel.lm, data = pHEire)
```

We use the same `panel.lm` function as in the previous section. (This requires copying and pasting it into the R console, if R has been shut down.) Because the variable `LOGAlt`, the logarithmically transformed altitude, is numeric, it is divided into a number of conditioning intervals, and, for each interval, pH is plotted against SDI. In addition, the data are segregated based on the Forested factor. The number and position of intervals for `LOGAlt` can be controlled with the `given.values` argument; see the `coplot` help file. Without this argument, the numeric variable is divided into six intervals overlapping by approximately 50. An easier approach may be using the `number` argument. Run this command:

```
> coplot(pH ~ SDI | LOGAlt * fForested,
      panel = panel.lm, data = pHEire, number = 2)
```

Compare the resulting coplot (which is not shown here) with that in Fig. 7.18; this one has fewer panels. The `number` argument can also be used if the coplot crashes due to an excessive number of panels.

7.7.3 Jazzing Up the Coplot*

This section is slightly more complicated (hence the asterisk in the title), and may be omitted upon first reading.

Figure 7.18 shows the relationship between pH versus SDI, altitude and forestation (and their interactions). To demonstrate what can be done, we produce the same coplot as that in Fig. 7.18, but with points of different colours depending on temperature. Temperatures above average are indicated by a light grey dot, and those below average are shown by a dark dot (obviously, red and blue dots would be better). Before this can be done, we need to use the following code to create a new variable containing the grey colours.

```
> pHEire$Temp2 <- cut(pHEire$Temperature, breaks = 2)
> pHEire$Temp2.num <- as.numeric(pHEire$Temp2)
```

The `cut` function separates the temperature data into two regimes, because we use `breaks = 2`. We encounter a problem in that the output, `Temp2`, is a factor, as can be seen from entering:

```
> cut(pHEire$Temperature, breaks = 2)
  [1] (1.89,7.4] (1.89,7.4] (1.89,7.4] (1.89,7.4]
  [5] (1.89,7.4] (1.89,7.4] (1.89,7.4] (1.89,7.4]
  [9] (1.89,7.4] (1.89,7.4] (1.89,7.4] (1.89,7.4]
 [13] (1.89,7.4] (1.89,7.4] (7.4,12.9] (1.89,7.4]
 ...
[197] (7.4,12.9] (7.4,12.9] (7.4,12.9] (7.4,12.9]
[201] (7.4,12.9] (7.4,12.9] (7.4,12.9] (7.4,12.9]
[205] (7.4,12.9]
Levels: (1.89,7.4] (7.4,12.9]
```

Each temperature value is allocated to either the class 1.89 – 7.4 (below average) or 7.4 – 12.9 (above average) degrees Celsius. A factor cannot be used for colours or greyscales; therefore we convert `Temp2` to a number, using the `as.numeric` function. As a result, `pHEire$Temp2.num` is a vector with values 1 and 2. We could have done this in Excel, but the `cut` function is more efficient. We are now ready to create the coplot in Fig. 7.19, using the following R code.

```
> coplot(pH ~ SDI | LOGAlt * fForested,
    panel = panel.lm, data = pHEire,
    number = 3, cex = 1.5, pch = 19,
    col = gray(pHEire$Temp2.num / 3))
```

It seems that high pH values were obtained for low SDI values with Forested = 2 (2 represents nonforested and 1 is forested) and above average temperature.

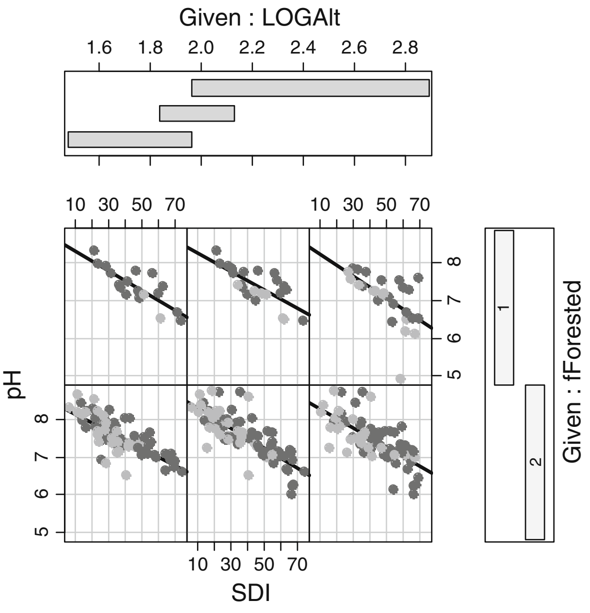


Fig. 7.19 Coplot for the pH data using four predictor variables: SDI, Forested, altitude, and temperature. The latter is shown by symbols of two shades of gray. *Light grey dots* correspond to above average temperature values, and *dark grey* are below average

7.8 Combining Types of Plots*

Here we touch upon R's more advanced graphing possibilities. There are several graphing systems that can be used in R. All the graphs we have shown were made by using the base package *graphics*. The R package called *grid* offers many advanced possibilities. It is possible to combine different plots into a single graph. We have already used the `mfrow` command to enable plotting several graphs on one screen. Here we use `layout` to create complex plot arrangements. Figure 7.20 shows a scatterplot of species richness versus NAP and also includes the boxplots of each variable.

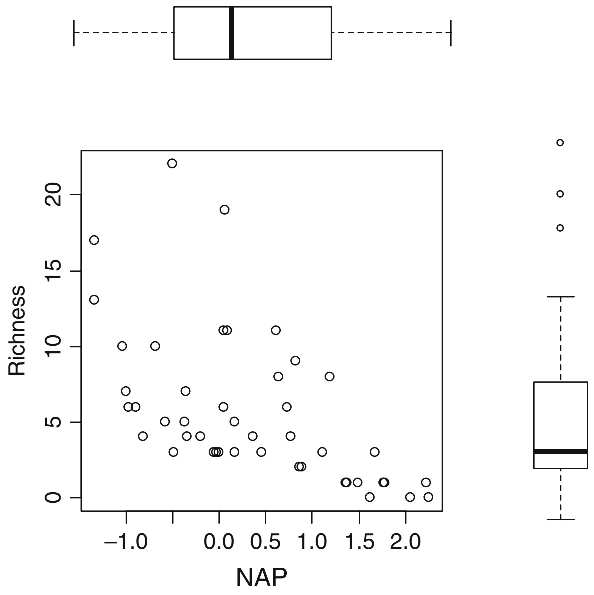


Fig. 7.20 Combination of scatterplot and boxplots for the benthic data

To produce this graph, we first need to define the number of graphs to incorporate, their placement, and their size. In this case, we want to arrange a 2-by-2 window with the scatterplot in the lower left panel, one of the boxplots in the upper left panel, and one boxplot in the lower right panel. For this we define a matrix, let's call it `MyLayOut`, with the following values.

```
> MyLayOut <- matrix(c(2, 0, 1, 3), nrow = 2, ncol=2,
                      byrow = TRUE)
> MyLayOut
     [,1] [,2]
[1,]    2    0
[2,]    1    3
```

The `matrix` command was introduced in Chapter 2. It looks intimidating, but simply creates a matrix with the elements 2 and 0 on the first row, and 1 and 3 on the second row. We use this matrix inside the `layout` function, followed by three plot commands. The first graph appears in the lower left corner (specified by the 1 in the matrix), the second plot in the upper left (specified by the 2), and the third graph in the lower right. Because there is a 0 in the upper right position of `MyLayout`, no graph will be drawn in that quadrant.

The next part of the code consists of

```
> nf <- layout(mat = MyLayOut, widths = c(3, 1),
        heights = c(1, 3), respect = TRUE)
```

The `widths` option specifies the relative width of the columns. In this case, the first column, containing the scatterplot and the boxplot for NAP, is 3, and the second column, containing the boxplot for richness, has a width of 1. The `heights` column specifies the height of the rows. The `respect = TRUE` ensures that a 1-unit in the vertical direction is the same as a 1-unit in the horizontal direction. The effect of these settings in the layout function can be visualised with the following command.

```
> layout.show(nf)
```

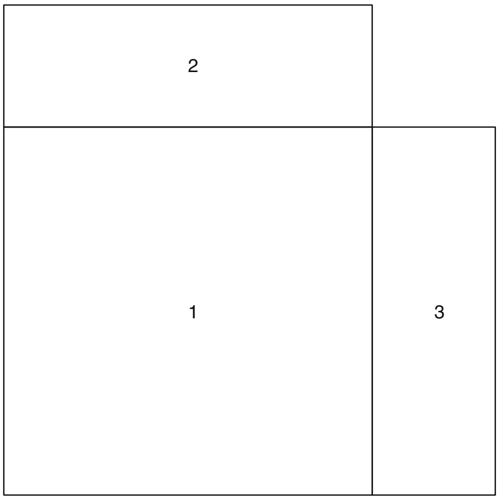


Fig. 7.21 Layout of the graphical window. The results of the first plot command will go into panel 1 (*lower left*), the next into panel 2, and the third plot into panel 3

All that remains is to make the three graphs. We must ensure that the range of the boxplot in panel 2 is synchronised with the range of the horizontal axis in panel 1, and the same holds for panel 3 and the vertical axis in panel 1. We also need to avoid excessive white space around the graphs, which means some trial and error with the `mar` values for each graph. We came up with the following code.

```
> xrange <- c(min(Benthic$NAP), max(Benthic$NAP))
> yrange <- c(min(Benthic$Richness),
              max(Benthic$Richness))
> #First graph
> par(mar = c(4, 4, 2, 2))
> plot(Benthic$NAP, Benthic$Richness, xlim = xrange,
       ylim = yrange, xlab = "NAP", ylab = "Richness")
> #Second graph
> par(mar = c(0, 3, 1, 1))
> boxplot(Benthic$NAP, horizontal = TRUE, axes = FALSE,
      frame.plot = FALSE, ylim = xrange, space = 0)
> #Third graph
> par(mar = c(3, 0, 1, 1))
> boxplot(Benthic$Richness, axes = FALSE,
          ylim = yrange, space = 0, horiz = TRUE)
```

Most of the options are self-explanatory. Change the values of the `mar`, and see what happens. Another function that can be used for similar purposes is the `split.screen`; see its help file.

7.9 Which R Functions Did We Learn?

Table 7.1 shows the R functions that were introduced in this chapter.

Table 7.1 R functions introduced in this chapter

Function	Purpose	Example
`pie`	Makes a pie chart	`pie(x)`
`pie3D`	Makes a 3-D piechart	`pie3D(x)`
`par`	Sets graph parameters	`par(...)`
`barplot`	Makes a bar chart	`barplot(x)`
`arrows`	Draws arrows	`arrows(x1,y1,x2,y2)`
`box`	Draws a box around the graph	`box()`
`boxplot`	Makes a boxplot	`boxplot(y)` `boxplot(y~x)`
`text`	Adds text to a graph	`text(x,y,"hello")`
`points`	Adds points to an existing graph	`points(x,y)`
`legend`	Adds a legend	`legend("topleft", MyText, lty = c(1,2,3))`

Table 7.1 (continued)

Function	Purpose	Example
`title`	Adds a title	`title(MyText)`
`expression`	Allows for special symbols	`ylab = expression(paste( delta^{15} , "N"))`
`pairs`	Creates multipanel scatterplots	`Pairs(X)`
`coplot`	Creates multipanel scatterplots	`Coplot(y~x|z)`
`layout`	Allows for multiple graphs in the same window	`layout(mat,widths,heights)` `plot(x)` `plot(y)`

7.10 Exercises

Exercise 1. The use of the `pie` function using the avian influenza data.
In Section 7.1, we used the total number of bird flu cases per year. Make a pie chart to illustrate the totals by country. Place the labels such that they are readable. The file *BirdFludeaths.txt* contains the data on deaths from the disease. Make a pie chart showing total deaths per year and one showing deaths per country.

Exercise 2. The use of the `barchart` and `stripchart` functions using a vegetation dataset.

In Section 4.1, we calculated species richness, as well as its mean values and standard deviations, in eight transects. Make a bar chart for the eight mean values and add a vertical line for the standard error.

Make a graph in which the means are plotted as black points, the standard errors as lines around the mean, and the observed data as open dots.

Exercise 3. The use of the `boxplot` function using a vegetation dataset.

Using the vegetation data in Exercise 2, make a boxplot showing the richness values.

Exercise 4. The use of the `boxplot` function using a parasite dataset.

In Section 6.3.3, a cod parasite dataset was used. Make a boxplot of the number of parasites (Intensity) conditional on area, sex, stage, or year. Try combinations to detect interactions.

Exercise 5. The use of the `dotchart` function using the owl data.

In Section 7.3, we used the owl data. Make two Cleveland dotplots of nestling negotiation and arrival time. Make a Cleveland dotplot showing arrival time per night. The nest and food treatment variables show which observations were made on the same night. See also Exercise 2 in Section 6.6.

Exercise 6. The use of the `dotchart` function using the parasite data.

Make a Cleveland dotplot for the parasite data that were used in Exercise 4. Use the number of parasites (Intensity), and group the observations by area, sex, stage, or by year. Make a Cleveland dotplot showing depth, and group the observations by prevalence.

Exercise 7. The use of the `plot` and `axis` functions using the owl data.

Apply a logarithmic transformation (use 10 as the base) on the nestling negotiation data. Add the value of 1 to avoid problems with the log of 0. Plot the transformed nestling negotiation data versus arrival time. Note that arrival time is coded as 23.00, 24.00, 25.00, 26.00, and so on. Instead of using the labels 25, 26, etc. for arrival time, use 01.00, 02.00, and so on.

Make the same graph, but use back-transformed values as labels along the vertical axis. This means using the log-transformed nestling negotiation data but with the label 1 if the log-transformed value is 0, 10 if the log-transformed value is 1, and so on.

Exercise 8. The use of the `legend` function using the owl data.

Add a smoother (see Chapter 5) to the graph created in Exercise 7 to visualise the pattern for the male data and for the female data. Extract the data from the males, fit a smoother, and superimpose this line onto the graph. Do the same for the female data. Use a legend to identify the different curves. Do the same for food treatment and night.

Exercise 9. The use of the `pairs` function using the vegetation data.

Make a pairplot for all the climatic variables in the vegetation data. Add correlation coefficients in the lower panels. What does the graph tell you?

Exercise 10. The use of the `coplot` function using the vegetation data.

Plot species richness versus a covariate of your choice conditional on transect.