Chapter 8
An Introduction to the Lattice Package

R contains many functions that can be used to draw graphs for specific types of data. Many of these plotting functions are not available in the packages that are loaded by default when R is started. Some are as simple to use as the `plot` function; others require more effort. The lattice package allows R to reach its full potential for imaging higher-dimensional data. We used one type of lattice plot, the multipanel scatterplot, in Section 1.4.1 to plot density of deep-sea pelagic bioluminescent organisms versus depth.

The lattice package was written by Deepayan Sarkar, who recently published an excellent book which we highly recommend (Sarkar, 2008). The package implements the Trellis Graphics framework developed at Bell Labs in the early 1990s.

In Chapter 7 we introduced the `coplot` function, which is particularly useful for displaying subsets of data in separate panels, when there is a grouping structure to the data. The lattice package allows taking this feature much further, but it comes at the price of more programming effort. However, by now you should have gained enough proficiency to master the function without too much difficulty.

8.1 High-Level Lattice Functions

The lattice user interface primarily consists of a number of generic functions called "high-level" functions, each designed to create a particular type of statistical display (Table 8.1). Fortunately, it is not necessary to learn each function individually. They are designed with a similar formula interface for different types of multipanel conditioning and respond to a large number of common arguments. Hence, once one function is mastered, learning to use the other functions is simple.

The plotting is performed by a default panel function embedded in each generic function that is applied to each panel. Most often the user will not be aware that a panel function is responding to arguments given in the function call. Names of default panel functions are generally self-explanatory. For example, the default

A.F. Zuur et al., *A Beginner's Guide to R*, Use R,
DOI 10.1007/978-0-387-93837-0_8, © Springer Science+Business Media, LLC 2009

panel function for the high-level function `histogram` is `panel.histogram`, for `densityplot` it is `panel.densityplot`, and for `xyplot` it is `panel.xyplot`, and so on. These predefined functions are available for you to use and to modify. We discuss panel functions more fully in Section 8.6.

Table 8.1 shows some of the high-level functions available in lattice.

Table 8.1 High-level functions of the lattice package

Function	Default Display
`histogram()`	Histogram
`densityplot()`	Kernel density plot
`qqmath()`	Theoretical quantile plot
`qq()`	Two-sample quantile plot
`stripplot()`	Stripchart (comparative 1-D scatterplots)
`bwplot()`	Comparative box-and-whisker plots
`dotplot()`	Cleveland dotplot
`barchart()`	Barplot
`xyplot()`	Scatterplot
`splom()`	Scatterplot matrix
`contourplot()`	Contour plot of surfaces
`levelplot()`	False colour level plot of surfaces
`wireframe()`	Three-dimensional perspective plot of surfaces
`cloud()`	Three-dimensional scatterplot
`parallel()`	Parallel coordinates plot

Do Exercise 1 in Section 8.11. This introduces lattice plots and provides an overview of the possibilities of the package.

8.2 Multipanel Scatterplots: `xyplot`

In the exercises in Chapter 4 we used temperature data measured at 30 stations along the Dutch coastline over a period of 15 years. Sampling took place 0 to 4 times per month, depending on the season. In addition to temperature, salinity was recorded at the same stations, and these measurements are used here. The data (in the file *RIKZENV.txt*) are submitted to the `xyplot` function to generate a multipanel scatterplot. The following is the code to enter the data into R, create a new variable, `MyTime`, representing time (in days), and create a multipanel scatterplot.

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

```
>Env$MyTime <- Env$Year + Env$dDay3 / 365
>library(lattice)

>xyplot(SAL ~ MyTime | factor(Station), type = "l",
      strip = function(bg, ...)
      strip.default(bg = 'white', ...),
 col.line = 1, data = Env)
```

The function xyplot contains characteristics common to all high-level lattice functions. The most obvious ones are the use of a formula, the vertical bar (also called the pipe symbol) within the formula, and the data argument. Lattice uses a formulalike structure that is also used in R for statistical models. The variable preceding the tilde (the ~) is plotted on the *y*-axis with that following along the *x*-axis. The conditioning variable (in this case Station), which will generate multiple panels, follows the vertical bar.

When there is no conditioning variable, the result of xyplot will be similar to the normal plot function; the data will be plotted in a single panel. The conditioning variable is usually a factor (note that we typed: factor (Station)), but it may also be a continuous variable. The default behaviour when using a continuous variable for conditioning is to treat each of its unique values as a discrete level. Often, however, the variable may contain so many values that it is advisable to partition it into intervals. This can be achieved by using the functions shingle and equal.count; see their help pages.

Figure 8.1 displays the data in five rows of 6 panels, showing a graph for each station. The station name is given in the horizontal bar, called the strip, above the panel.

The code is not difficult. The graph is drawn by the xyplot function using a formula to plot salinity versus (~) time, conditional on (|) station. We added two xyplot arguments: strip, used to create a white background in each strip, and col.line = 1 to designate black lines (recall from Chapter 5 that the colour 1 refers to black). The two other attributes, type and data, should be familiar; however, the type attribute in xyplot has more options than in the standard plot function. For example, type = " r" adds a regression line, type= " smooth " adds a LOESS fit, type = " g" adds a reference grid, type = " l" draws a line between the points, and type ="a" adds a line connecting the means of groups within a panel.

The strip argument should contain a logical (either TRUE or FALSE), meaning either do, or do not, draw strips, or a function giving the necessary input (in this case strip.default). To see what these options do, run the basic xyplot command.

```
> xyplot(SAL ~ MyTime |factor(Station), data = Env)
```

Compare this with (results are not shown here):

```
> xyplot(SAL ~ MyTime | factor(Station), type = "l",
        strip = TRUE, col.line = 1, data = Env)
```

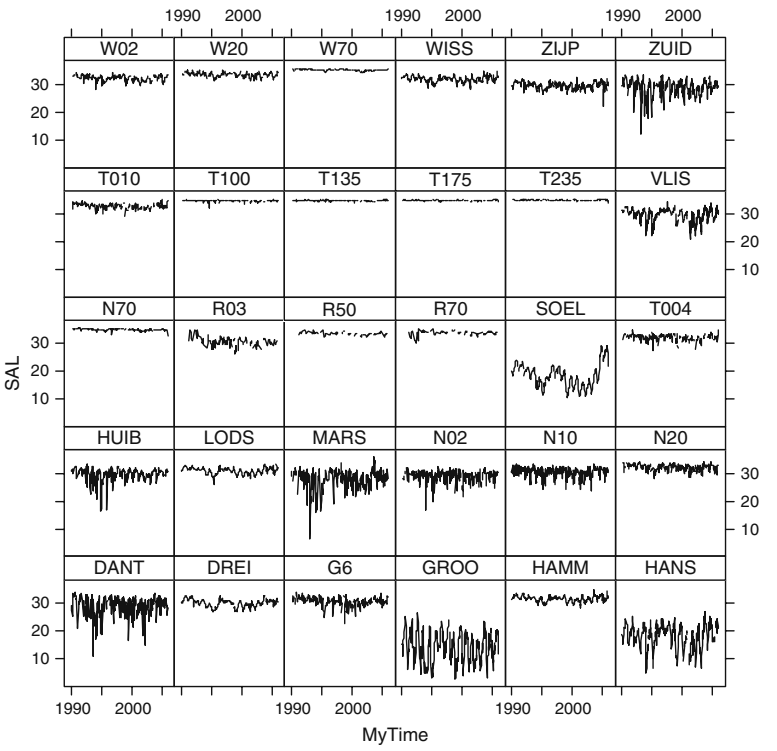

Fig. 8.1 Multipanel plot showing salinity (SAL) at 30 stations along the Dutch coast over 15 years. Note the differences among stations in spread and average values

```
> xyplot(SAL ~ MyTime | factor(Station), type = "l",
        strip = FALSE, col.line = 1, data = Env)
```

From the graph in Fig. 8.1 we note that some stations have generally lower salinity levels. Water in the North Sea has a salinity of around 32, probably because of proximity to rivers or other sources of fresh water inflow. Salinity values vary among stations, with stations that have lower values showing greater fluctuations over time. Another point to note is that some stations show similar patterns; possibly these stations are located near one another. It is difficult to see whether there is a seasonal pattern in these data. To investigate this, we can utilise the lattice function, `bwplot`, to draw box-and-whisker plots.

Do Exercise 2 in Section 8.11. This is an exercise in using the `xyplot` function with a temperature dataset.

8.3 Multipanel Boxplots: bwplot

A box-and-whisker plot, or boxplot, of the salinity data is shown in Fig. 8.2. The function boxplot was introduced in Chapter 7. The multipanel counterpart is called bwplot and uses a formula layout similar to the function xyplot. This time, however, we plot Salinity against Month (numbered 1–12) and our conditioning variable is not Station, but Area. There are two reasons for doing this. First, we have seen that some stations show similar patterns and we know that these are located in the same area. The second reason is that there are not sufficient data per station for each month to draw meaningful box-and-whisker plots, so we combine stations and years. Hence, the panels show the median and spread of the salinity data for each month in each of ten areas. Here is the code.

```
> setwd("C:/RBook")
> Env <- read.table(file ="RIKZENV.txt", header = TRUE)
> library(lattice)
> bwplot(SAL ~ factor(Month) | Area,
      strip = strip.custom(bg = 'white'),
      cex = 0.5, layout = c(2, 5),
      data = Env, xlab = "Month", ylab = "Salinity",
      par.settings = list(
        box.rectangle = list(col = 1),
        box.umbrella = list(col = 1),
        plot.symbol = list(cex = .5, col = 1)))
```

The code appears extensive but could have been shorter if we had not wanted to draw all items in the graph in black and white (by default colours are used). If colours and labels are not a consideration use (results are not presented here):

```
> bwplot(SAL ~ factor(Month) | Area, layout = c(2, 5),
         data = Env)
```

However, this graph is not as appealing, and we continue with the more extensive code. The list following par.settings is used to set the colour of the box, the whiskers (called umbrella), and the size and colour of the open circles (representing the median). We again set the strip colour to white. We use the layout argument to set the panel arrangement to a rectangular grid by entering a numeric vector specifying the number of columns and rows.

The variability in the data, as displayed in Fig. 8.2, differs among the areas. There also appears to be a cyclic component, probably illustrating a seasonal effect (e.g., river run-off); however, this is not equally clear for all areas.

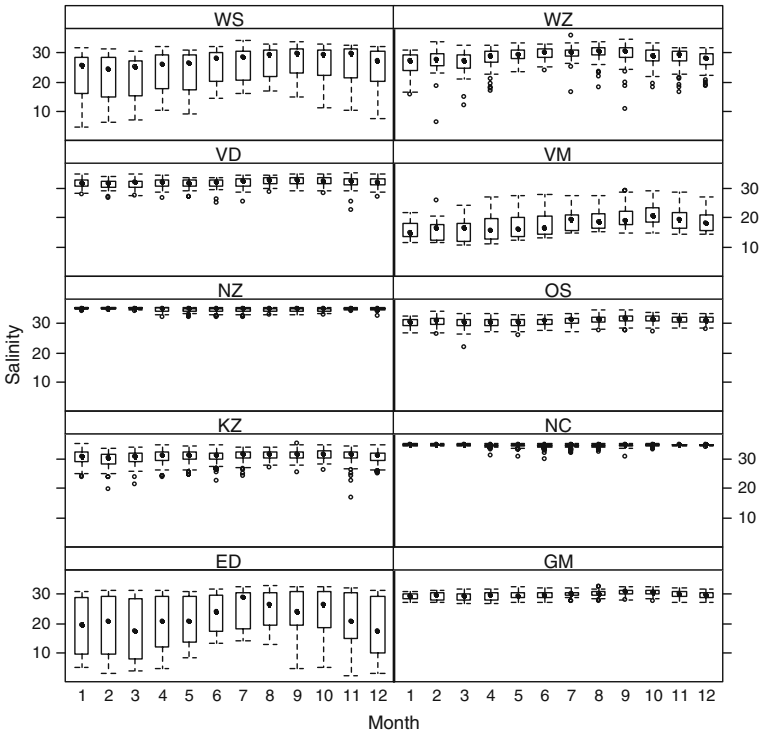

Fig. 8.2. Multipanel plot showing salinity for regions over time. Variability in salinity levels differs among regions

Do Exercise 3 in Section 8.11 in the use of the `bwplot` function using the temperature data.

8.4 Multipanel Cleveland Dotplots: `dotplot`

The Cleveland dotplot, called `dotplot` in lattice, was introduced in Chapter 7 as `dotchart`. Because there are so many data points in the salinity dataset, we restrict our plot to stations in a single area. The following code produces a multipanel dotplot, and the resulting graph is in Fig. 8.3.

```
> setwd("C:/RBook")
> Env <- read.table(file ="RIKZENV.txt", header = TRUE)
> library(lattice)
```

```
> dotplot(factor(Month) ~ SAL | Station,
    subset = Area=="OS", jitter.x = TRUE, col = 1,
    data = Env, strip = strip.custom(bg = 'white'),
    cex = 0.5, ylab = "Month", xlab = "Salinity")
```

The code is similar to that of the `xyplot` and `bwplot` functions. We reversed the order of salinity and month in the formula to ensure that salinity is plotted along the horizontal axis and month along the vertical axis (so that it matches the interpretation of the `dotchart` function; see Chapter 7).

There are two additional arguments in the code, `subset` and `jitter.x`. The `subset` option was used to create a subselection of the data. The OS stands for the area, Oosterschelde, and `jitter.x = TRUE` adds a small amount of random variation in the horizontal direction to show multiple observations with the same value in the same month.

Figure 8.3 shows data points that appear to be outside the normal range, potential outliers. It may be advisable to remove these before doing statistical analyses. However, this is a subjective choice that should not be made lightly.

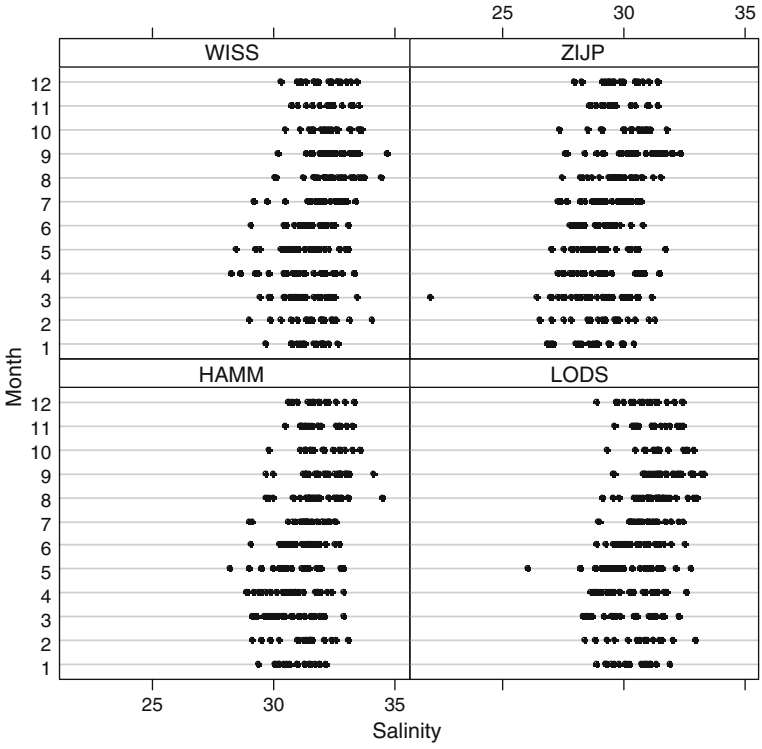

Fig. 8.3 Multipanel dotplot showing the salinity data for the four stations in the OS area. Each data point is displayed as a dot. The *y*-axis shows the month and the *x*-axis the salinity values. Note the two potential outliers in stations ZIJP and LODS

It is the responsibility of the owner of the data to make sure that data removal can be justified. It may be that the two low salinity values were the result of excessive rainfall. If the intent is to relate precipitation with salinity we might want to keep these data points.

Do Exercise 4 in Section 8.11 in the use of the multipanel `dotplot` function using temperature data.

8.5 Multipanel Histograms: `histogram`

The function `histogram` in the lattice package can be used to draw multiple histograms. The code below draws Fig. 8.4.

```
> setwd("C:/RBook")
> Env <- read.table(file ="RIKZENV.txt", header = TRUE)
> library(lattice)
> histogram( ~ SAL | Station, data = Env,
      subset = (Area == "OS"), layout = c(1, 4),
      nint = 30, xlab = "Salinity", strip = FALSE,
      strip.left = TRUE, ylab = "Frequencies")
```

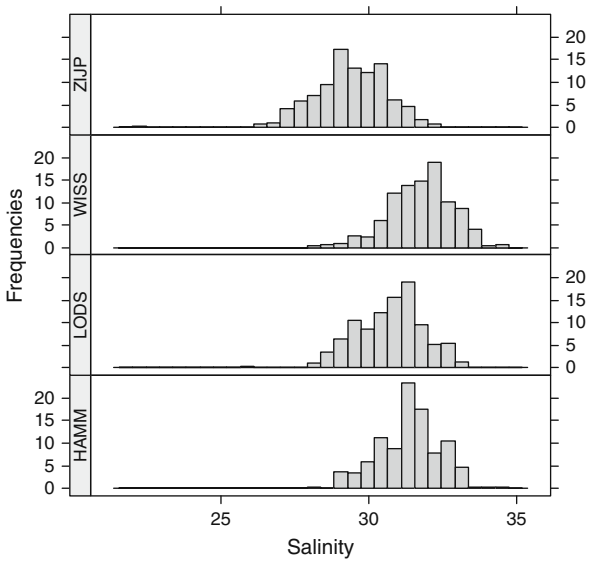

Fig. 8.4 A lattice `histogram` of salinity data for stations of the OS area

Note the slightly different format of the formula (we only need salinity data to plot the histograms). Again, we only present data for four stations using `subset`. We have changed the `layout` so that the panels are arranged vertically and increased the number of bars, the so-called bins, to 30 with the `nint` argument, as we found the default number to be too few. We also moved the strip to the side of the panels by setting `strip = FALSE` and `strip.left = TRUE`. Within the OS area there appears to be one station, ZIJP, with generally lower salinity.

To create a density plot, change the function name `histogram` to `densityplot`. If the argument for the number of bins is not removed, R will ignore it. Another function for plotting data distributions is `qqmath`, which draws QQ-plots. This stands for Quantile–Quantile plots, which are used to compare distributions of continuous data to a theoretical distribution (most often the Normal distribution).

8.6 Panel Functions

Panel functions were introduced in Chapter 7 with `pairs` and `coplot`. Remember that they are ordinary functions (see Chapter 6) that are used to draw the graph in more than one panel.

Panel functions in lattice are executed automatically within each high-level lattice function. As mentioned in Section 8.1, each default panel function contains the name of its "parent" function, for example, `panel.xyplot`, `panel.bwplot`, `panel.histogram`, and so on. Thus, when you type `xyplot(y ~ x | z)`, R executes: `xyplot(y ~ x | z, panel = panel.xyplot)`. The argument `panel` is used to associate a specific panel function with the plotting regime. Because a panel function is a function we could have written:

```
xyplot (y ~ x | z, panel = function (...) {
        panel.xyplot(...)})
```

The "`...`" argument is crucial, as it is used to pass on information to the other functions. Apart from `y, x,` and `z, xyplot` calculates a number of parameters before doing the actual plotting, and those that are not recognized are handed down to the `panel` function where they are used if requested. The consequence is that you can provide arguments to the panel functions at the level of the main function as well as within the panel function. You can write your own panel functions, but lattice contains a number of predefined functions that are easier to use. Panel functions can, and often do, call other panel functions, depending on the arguments. We discuss three examples of the use of panel functions.

8.6.1 First Panel Function Example

This example again uses the salinity dataset, this time to explore the potential relationship between rainfall and salinity. There are no precipitation data, so we

use Month as a continuous variable, assuming that rainfall is linked to time of year. We restrict the data to a single station (GROO) and condition this subset on Year. Within xyplot we call three panel functions: panel.xyplot, panel.grid, and panel.loess. We set limits for Month of 1–12 and for Salinity of 0–30.

```
> setwd("C:/RBook")
> Env <- read.table(file ="RIKZENV.txt", header = TRUE)
> library(lattice)
> xyplot(SAL ~ Month | Year, data = Env,
    type = c("p"), subset = (Station == "GROO"),
    xlim = c(0, 12), ylim = c(0, 30), pch = 19,
    panel = function (...){
    panel.xyplot(...)
    panel.grid(..., h = -1, v = -1)
    panel.loess(...) })
```

The resulting graph is presented in Fig. 8.5. Note how the points are on the gridlines. This is because panel.grid comes after panel.xyplot in the panel function. If you reverse the order of panel.grid and panel.xyplot, the grid is automatically drawn first. The panel function panel.loess adds a smoothing line. The amount of smoothing can be controlled by adding span = 0.9 (or any other value between 0 and 1) as a main attribute to xyplot (see Hastie and Tibshiranie (1990) for details on LOESS smoothing and span width).

We included options in the panel.grid function to align the vertical and horizontal gridlines with the axes labels. A positive number for h and v specifies the number of horizontal and vertical gridlines. If negative values for h and v are specified, R will try to align the grid with the axes labels. Experiment with different values for h and v, and see what happens.

Another important point is that, without including panel.xyplot in the code, the data points will not be plotted. Because Year is interpreted as a continuous variable, the strip has a different format than if Year were a factor. The year is represented by a coloured vertical bar in the strip. This is not very useful, and it is probably advisable to define year as a factor, so that it will print the values for year in the strips.

The data show clear signs of seasonality, although there is apparent variation in the annual salinity patterns. Nearly the same figure can be obtained with:

```
> xyplot(SAL ~ Month | Year, data = Env,
    subset = (Station == "GROO"), pch = 19,
    xlim = c(0, 12), ylim = c(0, 30),
    type = c("p", "g", "smooth"))
```

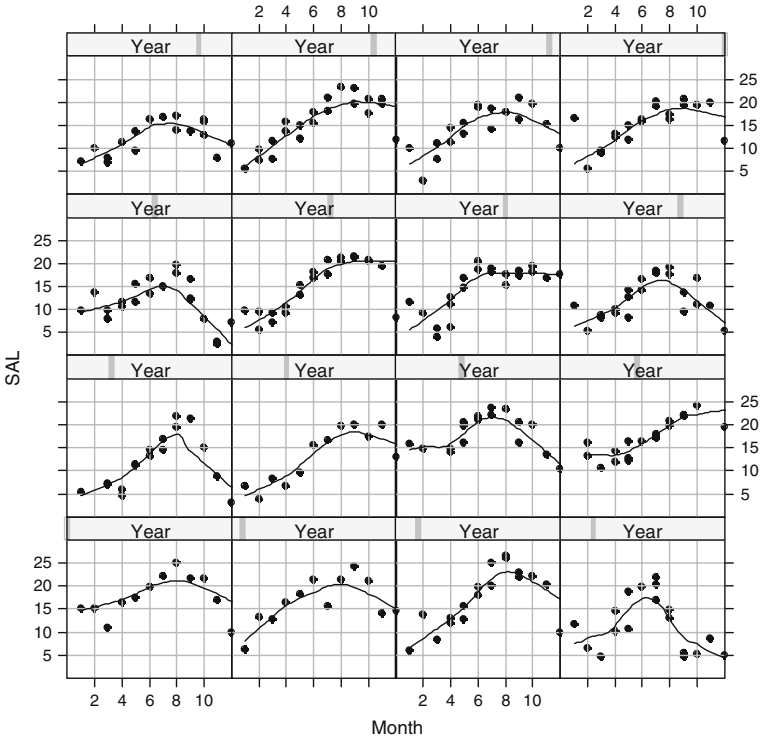

Fig. 8.5 Scatterplots of `Salinity` versus `Month` over the course of 16 years, with the addition of a *grid* and a *smoothing line*. The data show a clear seasonal pattern. Because Year is not defined as a factor, they are represented by *vertical lines* in the strips

Note that the `type` argument has the values `"p"`, `"g"` and `"smooth"`. As a result, the `xyplot` function executes the `panel.xyplot`, `panel.grid`, and `panel.smooth` functions.

8.6.2 Second Panel Function Example

The second example presents the multipanel Cleveland dotplot shown in Fig. 8.3, this time using a different colour and increased size for the dots representing potential outliers. The graph is shown in Fig. 8.6. Because this book is in greyscale, the two larger red points are printed in black.

Figure 8.6 can be created by two methods. The first option is to use the same code as in Section 8.4, and add the code `cex = MyCex` as the main argument, where `MyCex` is a vector of the same length as SAL with predefined values for `cex`. The second option is to determine values for `cex` in the panel function. The following demonstrates the second approach.

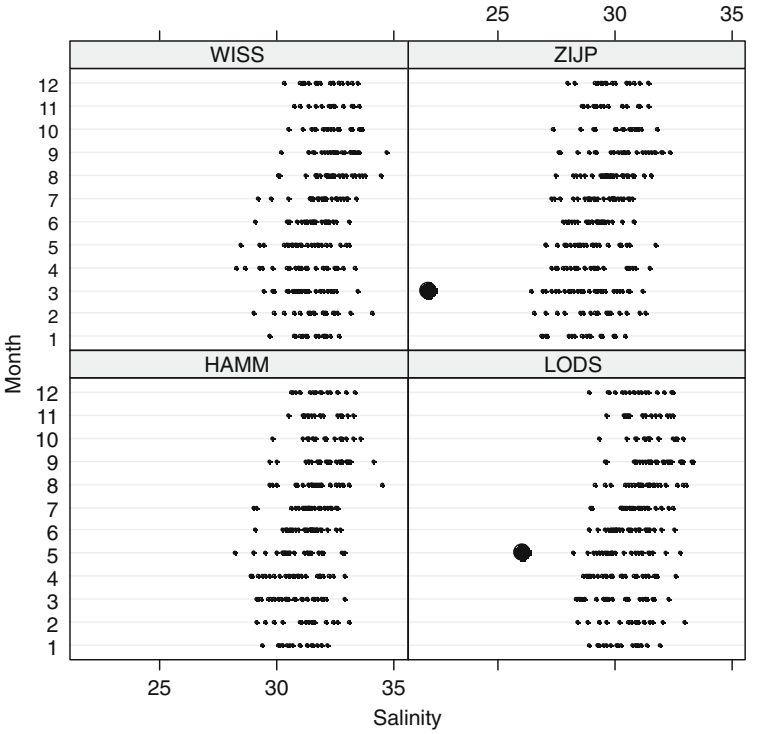

Fig. 8.6 Multipanel Cleveland dotplot with potential outliers shown by a larger dot size

A cut-off level for increasing the point size and changing its colour was set at salinity lower than the median minus three times the difference between the third and first quartiles. Note that this is a subjective cut-off level. The following code was used.

```
> setwd("C:/RBook")
> Env <- read.table(file ="RIKZENV.txt", header = TRUE)
> library(lattice)
> dotplot(factor(Month) ~ SAL | Station, pch = 16,
    subset = (Area=="OS"), data = Env,
    ylab = "Month", xlab = "Salinity",
    panel = function(x, y, ...) {
      Q <- quantile(x, c(0.25, 0.5, 0.75) ,
                    na.rm = TRUE)
      R <- Q[3] - Q[1]
      L <- Q[2] - 3 * (Q[3] - Q[1])
      MyCex <- rep(0.4, length(y))
      MyCol <- rep(1, length(y))
```

```
MyCex[x < L] <- 1.5
MyCol[x < L] <- 2
panel.dotplot(x, y, cex = MyCex,
              col = MyCol, ...)})
```

The main arguments are the `formula`, `data`, `xlab`, and `ylab`. The panel function has as arguments `x`, `y`, and "...". This means that inside the panel function, `x` contains the salinity data for a specific station, and `y` the corresponding months. Inside a panel, the `x` and the `y` constitute a subset of the data corresponding to a particular station. The "..." is used to pass on general settings such as the `pch` value. The `quantile` function is used to determine the first and third quantiles and the median. The cut-off level is specified (`L`), and all `x` values (salinity) smaller than `L` are plotted with `cex = 1.5` and `col = 2`. All other values have the values `cex = 0.4` and `col = 1`. The code can be further modified to identify considerably larger salinity values. In this case, `L` and `x < L` must be changed. We leave this as an exercise to the reader.

*8.6.3 Third Panel Function Example**

This section discusses graphing tools that can be used to illustrate the outcome of a principal component analysis (PCA). It is marked with an asterisk, as the material is slightly more difficult, not with respect to the R code, but due to the use of multivariate statistics. It is an exception in being one of the few parts of this book that requires knowledge of statistics to follow the text. If the graph in Fig. 8.7 looks interesting, read on.

Figure 8.7 shows four biplots.[1] The data used here are morphometric measurements taken on approximately 1000 sparrows (Chris Elphick, University of Connecticut, USA). Zuur et al. (2007) used these data to explain PCA in detail.

The interpretation of a PCA biplot depends on various choices, and a full discussion is outside the scope of this text. See Jolliffe (2002) or Zuur et al. (2007) for details. In this case, the morphometric variables are represented as lines from the origin to a point, with coordinates given by the loadings of the first two axes. The specimens are presented as points with coordinates given by the scores of the first two axes. Depending on the chosen scaling, loading and/or scores need to be multiplied by corresponding eigenvalues (Jolliffe, 2002).

The biplot allows us to make statements of which variables are correlated, which specimens are similar, and whether specimens show high (or low) values

[1] A biplot is a tool to visualise the results of multivariate techniques such as principal component analysis, correspondence analysis, redundancy analysis, or canonical correspondence analysis. Using specific rules in the PCA biplot, correlations (or covariances) among the original variables, relationships among observations, and relationships between observations and variables can be inferred. There are various ways to scale the biplot, and the interpretation of the biplot depends on this scaling.

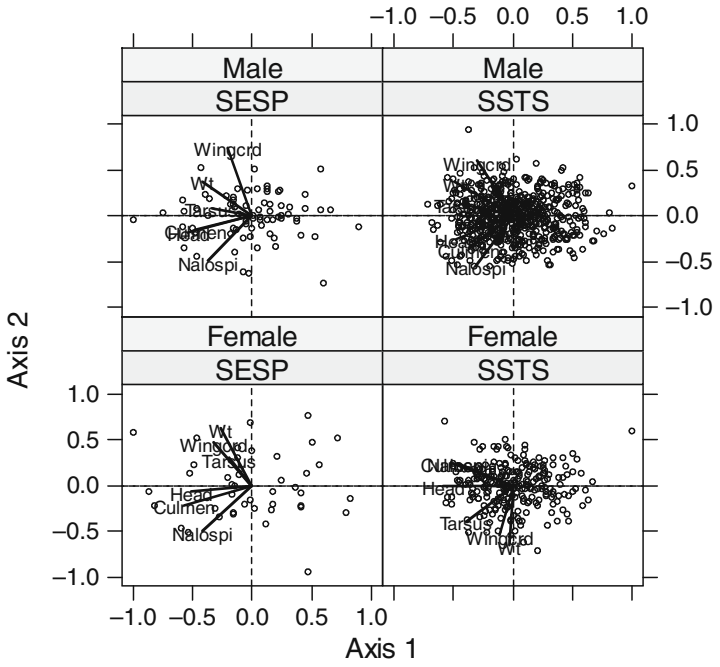

Fig. 8.7 Multipanel principal component analysis biplots. Each panel shows a biplot obtained by applying PCA (using the correlation matrix) to a dataset. SESP and SSTS represent the species seaside sparrows (*Ammodramus maritimus*) and saltmarsh sharp-tailed sparrows (*Ammodramus caudacutus*). The *graphs* indicate that the nalospi, culmen, and head measurements are correlated, and this makes sense as these are all nested subsets of each other. Wing length, mass, and tarsus on the other hand are indicators of the overall structure size of the bird, so again it makes sense that these are correlated (as suggested by the biplots), but not necessarily correlated with the first three

for particular variables. These statements are based on the directions of the lines and positions of the points. Lines pointing in a similar direction correspond to positively correlated variables, lines with an angle of 90 degrees correspond to variables that have a small correlation, and lines pointing in (approximately) opposite directions correspond to negatively correlated variables. There are also criteria for comparing points and comparing the points to the lines. The interested reader is referred to the aforementioned literature.

The sampled sparrows can be separated into two sexes and two species (SESP and SSTS). The following code was used to create Fig. 8.7.

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

```
> library(lattice)
> xyplot(Wingcrd ~ Tarsus | Species * Sex,
      xlab = "Axis 1", ylab = "Axis 2", data = Sparrows,
      xlim = c(-1.1, 1.1), ylim = c(-1.1, 1.1),
      panel = function(subscripts, ...){
        zi <- Sparrows[subscripts, 3:8]
        di <- princomp(zi, cor = TRUE)
        Load <- di$loadings[, 1:2]
        Scor <- di$scores[, 1:2]
        panel.abline(a = 0, b = 0, lty = 2, col = 1)
        panel.abline(h = 0, v = 0, lty = 2, col = 1)
        for (i in 1:6){
            llines(c(0, Load[i, 1]), c(0, Load[i, 2]),
                   col = 1, lwd = 2)
            ltext(Load[i, 1], Load[i, 2],
                  rownames(Load)[i], cex = 0.7)}
        sc.max <- max(abs(Scor))
        Scor <- Scor / sc.max
        panel.points(Scor[, 1], Scor[, 2], pch = 1,
                     cex = 0.5, col = 1)
       })
```

The `xlab`, `ylab`, and `data` arguments are familiar. The first part of the equation, `Wingcrd ~ Tarsus`, is used to set up the graph. There was no specific reason for choosing to use these two variables in the formula. The portion of the code following the | symbol is new. So far, we have only used one conditioning variable, but in this case there are two, `Species` and `Sex`. As a result, the lower two panels in the graph show the female data, and the upper two panels show the data from males. Change the order of `Species` and `Sex` to see what happens. Note that both variables are defined as characters in the data file; hence R automatically treats them as factors.

The `xlim` and `ylim` values require some statistical explanation. The outcome of a PCA can be scaled so that its numerical information (scores) for a graph fits between −1 and 1. See Legendre and Legendre (1998) for mathematical details.

It is also important when constructing the graph to ensure that the distance in the vertical direction is the same as in the horizontal direction to avoid distortion of the angles between lines.

We now address the more difficult aspect, the panel function. The vector `subscripts` automatically contains the row number of the selected data in the panel function. This allows us to manually extract the data that are being used for a certain panel using `Sparrows[ subscripts, 3:8]` . The `3:8` designates the variables `Wingcrd`, `Tarsus`, `Head`, `Culmen`, `Nalospi`,

and Wt.[2] The princomp function applies principal component analysis, and loadings and scores of the first two axes are extracted. The two panel.abline functions draw the axes through the origin. The loop (see Chapter 6) is used to draw the lines and to add the labels for all variables. The functions llines and ltext do the work. Finally, we rescale all the scores between −1 and 1, and add them as points with the panel.xyplot function.

The resulting biplots show how the correlations among the morphometric variables differ according to sex and species.

The code can easily be extended to allow for triplots obtained by redundancy analysis or canonical correspondence analysis (see the functions rda and cca in the package vegan).

Full details on PCA and biplots and triplots can be found in Jolliffe (2002), Legendre and Legendre (1998), and Zuur et al. (2007), among many other sources.

Do Exercise 6 in Section 8.11 on the use of panel functions using the temperature data.

8.7 3-D Scatterplots and Surface and Contour Plots

Plots for displays of three variables, sometimes called trivariate displays, can be generated by the functions cloud, levelplot, contourplot, and wireframe. In our opinion, three-dimensional scatterplots are not always useful. But they look impressive, and we briefly discuss their creation. Below is an example of the function cloud, applied to the Dutch environmental dataset, which produces three-dimensional scatterplots showing the relationships among chlorophyll-a, salinity, and temperature. The code is fairly simple, and the resulting graph is presented in Fig. 8.8.

```
> setwd("C:/RBook")
> Env <- read.table(file ="RIKZENV.txt", header = TRUE)
> library(lattice)
> cloud(CHLFa ~ T * SAL | Station, data = Env,
    screen = list(z = 105, x = -70),
    ylab = "Sal.", xlab = "T", zlab = "Chl. a",
    ylim = c(26, 33), subset = (Area=="OS"),
    scales = list(arrows = FALSE))
```

[2] In Chapter 2 we provided an explanation for Wingcrd, Tarsus, Head, and Wt. Culmen measures the length of the top of the bill from the tip to where feathering starts, and Nalospi the distance from the bill top to the nostril.

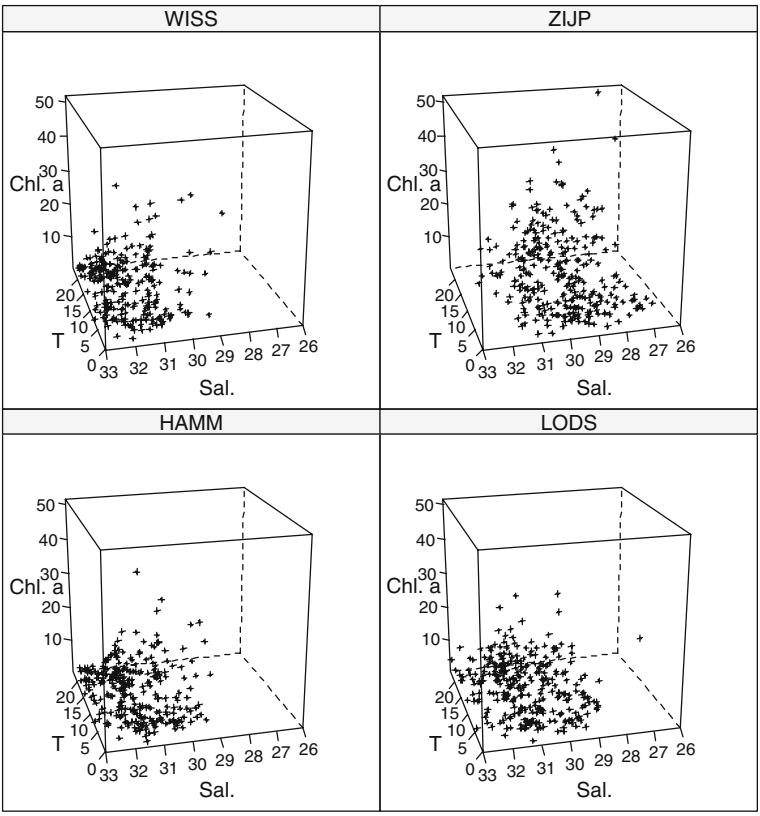

Fig. 8.8 Three-dimensional scatterplot of chlorophyll-a, salinity, and temperature

The function `cloud` uses several arguments that we have not previously introduced. The option `screen` is used to denote the amount of rotation about the axes in degrees. With `arrows = FALSE` we removed arrows that are normally plotted along the axes of three-dimensional graphs to indicate the direction in which values increase. Consequently, axis tick marks, which by default are absent, are now shown. We limited the *y*-axis values to between 26 and 33.

The functions `levelplot, contourplot`, and `wireframe` are used to plot surfaces. This generally involves predicting values on a regular grid by statistical functions, which is outside the scope of this book. More information on these functions can be found in their help pages.

8.8 Frequently Asked Questions

There are a number of things that we have often found ourselves modifying when making lattice plots. The following are some that we have found useful.

8.8.1 How to Change the Panel Order?

By default panels are drawn starting from the lower-left corner, proceeding to the right, and then up. This sequence can be changed by setting `as.table = TRUE` in a high-level lattice call, resulting in panels being drawn from the upper-left corner, going right, and then down.

The order of the panels can also be changed by defining the condition variable as a factor, and changing the `level` option in the `factor` function. Figure 8.9 shows a multipanel scatterplot of abundance of three bird species on three islands in Hawaii. The data were analysed in Reed et al. (2007). The problem with the graph is that the time series are arranged randomly with respect to species and island, which makes comparisons among time series of bird abundance of an island, or of an individual species, more difficult.

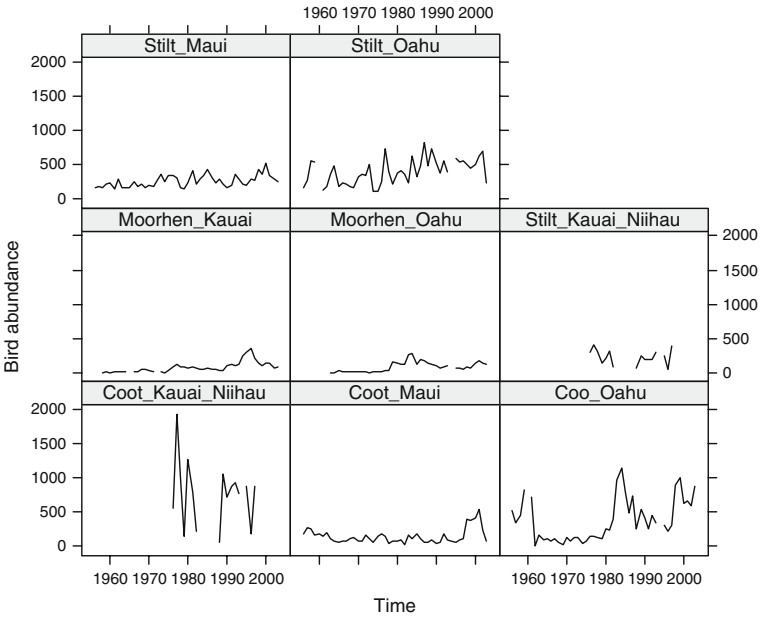

Fig. 8.9 Time series abundances of three bird species on three islands of Hawaii

Figure 8.10 on the other hand, shows the time series of each island in rows, and time series of each species in the columns. This makes the comparison of trends with respect to individual species or islands much easier. So, how did we do it?

The following code imports the data and uses the `as.matrix` and `as.vector` commands to concatenate the eight abundance time series into a single long vector. The `as.matrix` command converts the data frame into a matrix, which allows `as.vector` to make the conversion to a long vector; `as.vector` will not work with a data frame. The `rep` function is used to create a single long vector containing eight repetitions of the variable Year.

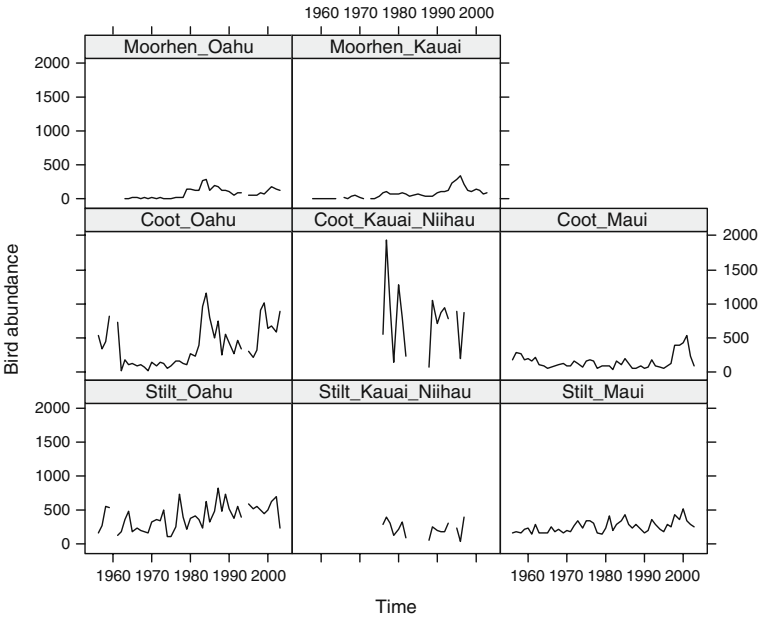

Fig. 8.10 Time series of abundance of three bird species on three islands of Hawaii. Note that time series of an island are arranged *vertically*, and time series of a species are *horizontal*

```
> setwd("C:/RBook")
> Hawaii <- read.table("waterbirdislandseries.txt",
                       header = TRUE)
> library(lattice)
> Birds <- as.vector(as.matrix(Hawaii[, 2:9]))
> Time <- rep(Hawaii$Year, 8)
> MyNames <- c("Stilt_Oahu", "Stilt_Maui",
               "Stilt_Kauai_Niihau","Coot_Oahu",
               "Coot_Maui", "Coot_Kauai_Niihau",
               "Moorhen_Oahu","Moorhen_Kauai")
> ID <- rep(MyNames, each = 48)
```

The `rep` function is also used to define a single long vector `ID` in which each name is repeated 48 times, as each time series is of length 48 years (see Chapter 2). Figure 8.9 was made with the familiar code:

```
> xyplot(Birds ~ Time | ID, ylab = "Bird abundance",
         layout = c(3, 3), type = "l", col = 1)
```

The layout option tells R to put the panels in 3 rows and 3 columns with points connected by a black line.

To change the order of the panels, change the order of the levels of the factor `ID`:

```
> ID2 <- factor(ID, levels = c("Stilt_Oahu",
      "Stilt_Kauai_Niihau", "Stilt_Maui",
      "Coot_Oahu", "Coot_Kauai_Niihau", "Coot_Maui",
      "Moorhen_Oahu", "Moorhen_Kauai"))
```

Note the change in the order of the names. Rerunning the same `xyplot` command, but with `ID` replaced by `ID2`, produces Fig. 8.10. Determining the order of the levels within the factor `ID2` (the names of the bird/island combinations) is a matter of trial and error.

8.8.2 How to Change Axes Limits and Tick Marks?

The most direct way to influence the range of values on the axes is by using `xlim` and `ylim`; however, this will result in the same limits on both the x and y axes of all panels. The `scales` option is more versatile. It can be used to define the number of tick marks, the position and labels of ticks, and also the scale of individual panels.

In Figure 8.10 the vertical ranges of the time series differ among the panels. This is obviously because some species are more abundant than others. However, if we want to compare trends over time, we are less interested in the absolute values. One option is to standardise each time series. Alternatively, we can allow each panel to set its own range limits on the y-axis. This is done as follows (after entering the code from the previous subsection).

```
> xyplot(Birds ~ Time|ID2, ylab = "Bird abundance",
      layout = c(3, 3), type = "l", col = 1,
      scales = list(x = list(relation = "same"),
                    y = list(relation = "free")))
```

The option `scales` can contain a list that determines attributes of both axes. In this case, it specifies that the x-axes of all panels have the same range, but sets a vertical range of each panel appropriate to the data. The resulting graph is presented in Fig. 8.11.

To change the direction of the tick marks inwards use the following code.

```
> xyplot(Birds ~ Time|ID2, ylab = "Bird abundance",
      layout = c(3, 3), type = "l", col = 1,
      scales = list(x = list(relation = "same"),
                    y = list(relation = "free"),
                    tck = -1))
```

The `tck = -1` is within the list argument of the scales option. There are many more arguments for scales; see the `xyplot` help file.

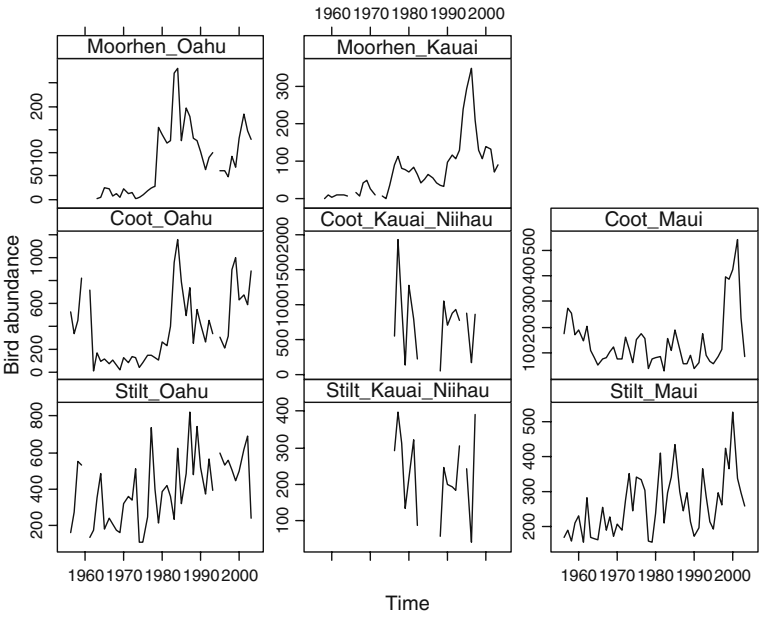

Fig. 8.11 Time series of abundance of three bird species on three islands of Hawaii. Each panel has an appropriate abundance value range

8.8.3 Multiple Graph Lines in a Single Panel

The attribute `groups` in high-level lattice functions can be used when there is a grouping in the data that is present in each level of the conditioning variable. Figure 8.12 shows all time series of a species in a single panel. The following code was used to generate the graph.

```
> Species <- rep(c("Stilt", "Stilt", "Stilt",
                   "Coot", "Coot", "Coot",
                   "Moorhen", "Moorhen"), each = 48)
> xyplot(Birds ~ Time | Species,
       ylab = "Bird abundance",
       layout = c(2, 2), type = "l", col = 1,
       scales = list(x = list(relation = "same"),
                     y = list(relation = "free")),
       groups = ID, lwd = c(1, 2, 3))
```

The first command defines a vector `Species` identifying which observations are from which species. The `xyplot` with the `groups` option then draws the time series of each species in a single panel. The option `lwd` was used to draw lines of different thickness to represent the three islands.

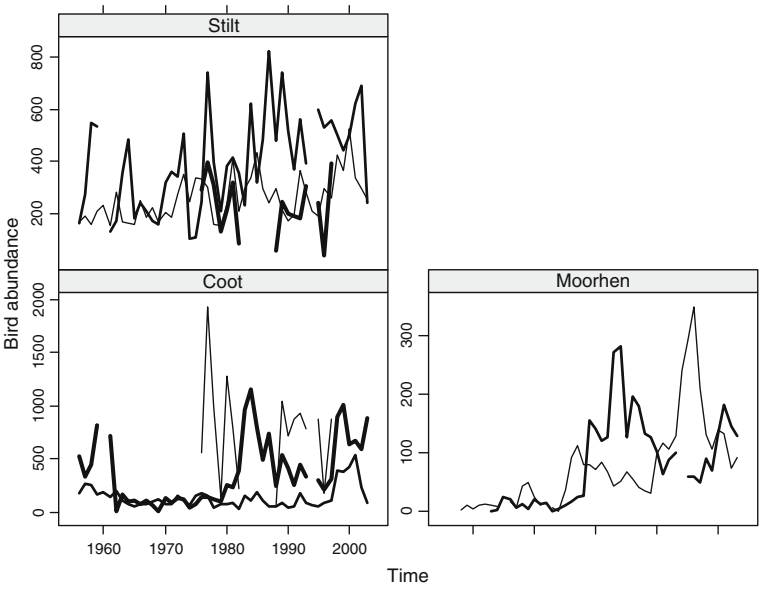

Fig. 8.12 Abundance of three species of Hawaiian birds over time. Data on each species are plotted in a single panel

Do Exercise 7 in Section 8.11 on creating multiple lines in an `xyplot` function using the temperature data.

*8.8.4 Plotting from Within a Loop**

If you did not read Chapter 6, you may want to skip this section. Recall from Section 8.2, that the salinity dataset consists of time series at stations, and that the stations are located in areas along the Dutch coast. In Section 8.6.2, we created a dotplot of the data from the area OS (Figure 8.6). Suppose we want to make the same graph for each of the 12 areas. One option is to enter the code from Section 8.6.2 12 times and each time to change the `subset` option. However, in Chapter 6, we demonstrated executing similar plotting commands automatically within a loop. The only difference is that we need to replace the `plot` command by a `dotplot` command:

```
> setwd("C:/RBook")
> Env <- read.table(file ="RIKZENV.txt", header = TRUE)
> library(lattice)
> AllAreas <- levels(unique(Env$Area))
> for (i in AllAreas ){
```

```
Env.i <- Env[Env$Area == i,]
win.graph( )
dotplot(factor(Month)~SAL | Station, data = Env.i)
}
```

The first three lines load the data and the lattice package. The variable `AllAreas` contains the names of the 12 areas. The loop iteratively extracts the data of an area and draws a dotplot of all stations in this area. The only problem is that this code will produce 12 empty graph windows.

When you execute a high-level lattice function, a graph is created on your screen. This appears similar to using, for example, the traditional `plot` command. The lattice command differs, however, because this command returns an object of class "trellis," and, in order to see a plot, the `print` function is invoked. Sometimes, when issuing a command to draw a lattice plot, nothing happens, not even an error message. This most often happens when creating a lattice plot inside a loop or function, or from a `source` command. To get the graphs, the `print` command must be embedded in the loop:

```
print(dotplot(factor(Month)~SAL | Station,
       data = Env.i))
```

Adding the `print` command to the code and rerunning it will produce 12 windows with graphs.

8.8.5 Updating a Plot

Because drawing lattice plots is time consuming, especially when you are new to lattice, the `update` function is useful. Many attributes of a lattice object can be changed with `update`, thus your graph must be stored in an object first. An additional advantage is that if you experiment by using the `update` command, your original graph is not changed, so

```
> MyPlot <- xyplot(SAL ~ MyTime | Station,
                   type = "l", data = Env)
> print(MyPlot)
> update(MyPlot, layout = c(10, 3))
```

will print the plot in a new layout. The `update` command will automatically generate a plot because it is not assigned to an object. The original object `MyPlot` is unchanged.

8.9 Where to Go from Here?

After completing the exercises you will have the flavour of lattice graphs and will undoubtedly want to use them in research, publications, and presentations. For further information consult Sarkar (2008) or Murrell (2006). Other

sources are the website that accompanies Sarkar (2008) (http://lmdvr.r-forge.r-project.org) or the R-help mailing list.

8.10 Which R Functions Did We Learn?

Table 8.2 contains the R functions introduced in this chapter.

Table 8.2 R functions introduced in this chapter

Function	Purpose	Example
`xyplot`	Draws a scatterplot	`xyplot (y ~ x | g, data = data)`
`histogram`	Histogram	`histogram(~ x | g, data = data)`
`bwplot`	Comparative box-and-whisker plots	`bwplot(y ~ x | g, data = data)`
`dotplot`	Cleveland dotplot	`dotplot(y ~ x | g, data = data)`
`cloud`	Three-dimensional scatterplot	`cloud(z ~ x * y | g, data = data)`

8.11 Exercises

Exercise 1. Using the `demo(lattice)` function.
Load the lattice package and investigate some of the possibilities by typing in `demo(lattice)`. Type in `?xyplot` and copy and paste some of the examples.

Exercise 2. Using the `xyplot` with temperature data.
Create a multipanel scatterplot in which temperature is plotted versus time for each station. What is immediately obvious? Do the same for each area. What goes wrong and how can you solve this? Add a smoother and a grid to each panel.

Exercise 3. Using the `bwplot` with temperature data.
Create a boxplot in which temperature is plotted versus month for each area. Compare with the boxplot for the salinity data and comment on the differences in the patterns.

Exercise 4. Using the `dotplot` function with salinity data.
Use Cleveland dotplots to discover if there are more outliers in the salinity data, making a lattice plot with all stations as panels. Compare with Fig. 8.3. What can be noted on the scale of the y-axis? Look up the argument `relation` in the help page of `xyplot` and use it.

Exercise 5. Using the density plot with salinity data.
Change Fig. 8.4 to a density plot. Is it an improvement? Add the following argument: `plot.points = "rug"`. To compare density distributions you might prefer to have all the lines in a single graph. This is accomplished with the

`groups` argument. Remove the conditioning argument and add `groups = Station` (see also Section 8.8). Add a legend to specify the lines representing each station. This requires advanced programming (though there are simple solutions), and we refer you to the code on our website for the solution.

Exercise 6. Using the `xyplot` function with temperature data.

Look at the help pages of `panel.linejoin`. Create a plot similar to Fig. 8.2, but with temperature on the *y*-axis. This is the same as in Exercise 3, but now use `panel.linejoin` to connect the medians, not the means. Take care of the NAs in the data, otherwise nothing will happen.

Exercise 7. Using the `xyplot` function with salinity data.

Create a lattice scatterplot using salinity as the dependent variable versus time for each area and include the `groups` argument to draw separate lines for each station.

Exercise 8. Using the `xyplot` function with temperature data.

In Exercise 2 you created a lattice scatterplot for each area using temperature as the dependent variable versus time. Make a similar graph for the area "KZ", but plot small dots and add a smoothing line with 1/10 span width. Create strips on either side of the panels, with the text "Area 1", "Area 2", and so on. Add an *x*-label, a *y*-label, and a title.

Exercise 9. Using the `xyplot` function with salinity data.

Create a multipanel scatterplot of the salinity data versus time conditional on area with different lines (no points) for the different stations within each area. Make sure the panel layout is in two columns. Use the same *x*-axis on each panel, but different scales for the *y*-axes. Limit the number of tick marks and labels on the *y*-axes to three or four and on the *x*-axes to four, with the tick marks between labels. Remove the tick marks (and labels) from the top and make sure they are only present on the bottom of the graph. Decrease the size of the text in the strip and the height of the strip. Add a grid (properly aligned with the tick marks), and also *x*- and *y*-labels. Change the order of the panels to alphabetic from top left to bottom right.

Exercise 10. Using the `xyplot` function with the ISIT data.

Create a multipanel scatterplot of the ISIST data (see Chapter 1). Plot the sources versus depth for each station. Also make a multipanel graph in which data from all stations sampled in the same season are grouped (see also Exercise 4 in Section 3.7). Each panel (representing a season) should have multiple lines.