5

Graphics

Producing high-quality graphics is one of the main reasons for doing statistical computing. The particular plot function you need will depend on the number of variables you want to plot and the pattern you wish to highlight. The plotting functions in this chapter are dealt with under four headings:

- plots with two variables;
- plots for a single sample;
- multivariate plots;
- special plots for particular purposes.

Changes to the detailed look of the graphs are dealt with in Chapter 29.

5.1 Plots with two variables

With two variables (typically the *response variable* on the *y* axis and the *explanatory variable* on the *x* axis), the kind of plot you should produce depends upon the nature of your explanatory variable. When the explanatory variable is a continuous variable, such as length or weight or altitude, then the appropriate plot is a **scatterplot**. In cases where the explanatory variable is categorical, such as genotype or colour or gender, then the appropriate plot is either a **box-and-whisker plot** (when you want to show the scatter in the raw data) or a **barplot** (when you want to emphasize the effect sizes).

The most frequently used plotting functions for two variables in R are the following:

- `plot(x,y)` scatterplot of *y* against *x*;
- `plot(factor, y)` box-and-whisker plot of *y* at each factor level;
- `barplot(y)` heights from a vector of *y* values (one bar per factor level).

The R Book, Second Edition. Michael J. Crawley.

5.2 Plotting with two continuous explanatory variables: Scatterplots

The `plot` function draws axes and adds a scatterplot of points. Two extra functions, `points` and `lines`, add extra points or lines to an existing plot. There are two ways of specifying `plot`, `points` and `lines` and you should choose whichever you prefer:

- Cartesian `plot(x,y)`
- formula `plot(y~x)`

The advantage of the formula-based plot is that the plot function and the model fit look and feel the same (response variable, tilde, explanatory variable). If you use Cartesian plots (eastings first, then northings, like the grid reference on a map) then the plot has '*x* then *y*' while the model has '*y* then *x*'.

At its most basic, the `plot` function needs only two arguments: first the name of the explanatory variable (`x` in this case), and second the name of the response variable (`y` in this case): `plot(x,y)`. The data we want to plot are read into R from a file:

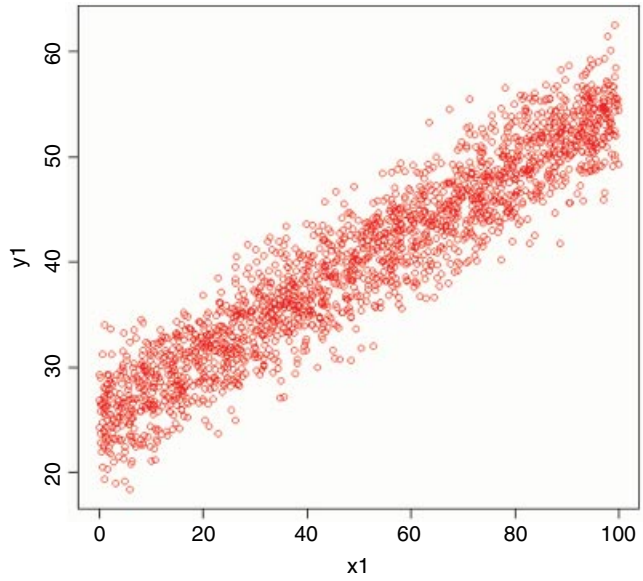
```
data1 <- read.table("c:\\temp\\scatter1.txt",header=T)
attach(data1)
names(data1)
```

```
[1] "x1" "y1"
```

Producing the scatterplot could not be simpler: just type

```
plot(x1,y1,col="red")
```

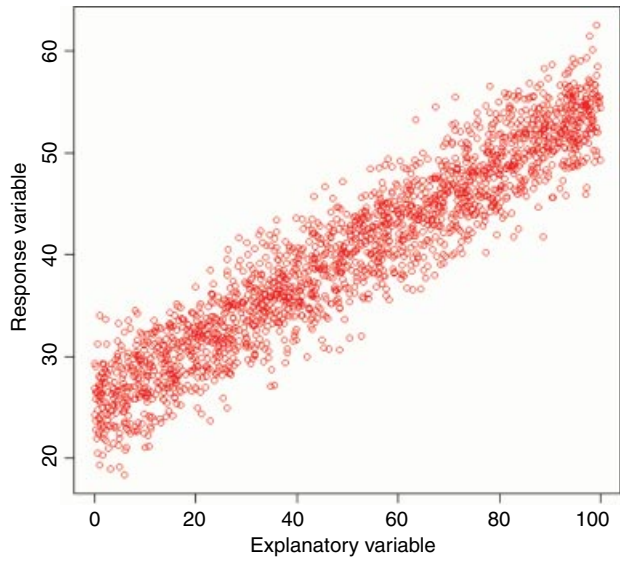
with the vector of *x* values first, then the vector of *y* values (changing the colour of the points is optional).



Notice that the axes are labelled with the variable names, unless you chose to override these with `xlab` and `ylab`. It is often a good idea to have longer, more explicit labels for the axes than are provided by the variable

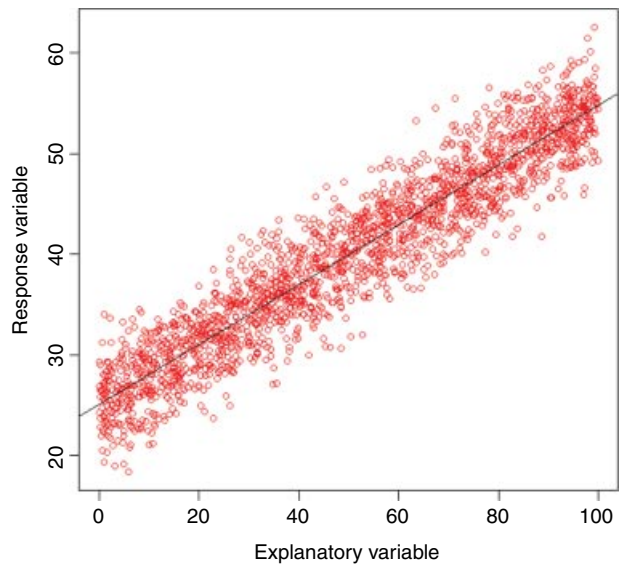
names that are used as default options (`x1` and `y1` in this case). Suppose we want to change the label `x1` into the longer label 'Explanatory variable' and the label on the *y* axis from `y1` to 'Response variable'. Then we use `xlab` and `ylab` like this:

```
plot(x1,y1,col="red",xlab="Explanatory variable",ylab="Response
variable")
```



The great thing about graphics in R is that it is extremely straightforward to add things to your plots. In the present case, we might want to add a regression line through the cloud of data points. The function for this is `abline` which can take as its argument the linear model object `lm(y1~x1)` as explained on p. 491:

```
abline(lm(y1~x1))
```



Just as it is easy to add lines to the plot, so it is straightforward to add more points. The extra points are in another file:

```
data2 <- read.table("c:\\temp\\scatter2.txt",header=T)
attach(data2)
names(data2)
```

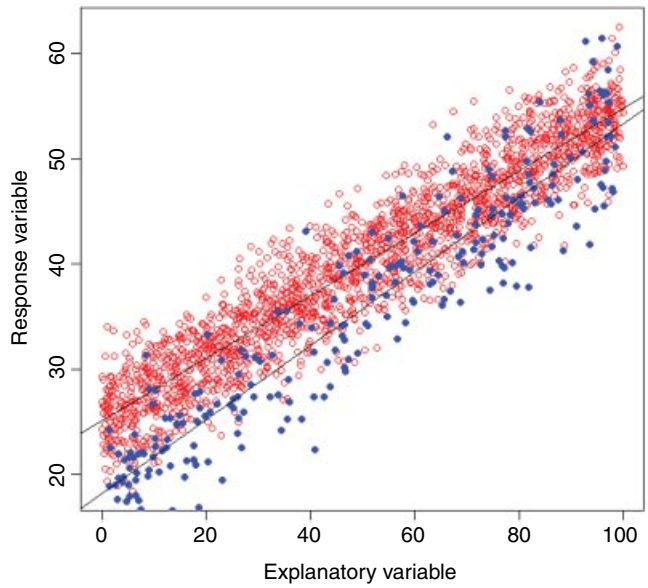
```
[1] "x2" "y2"
```

The new points (`x2,y2`) are added using the `points` function like this:

```
points(x2,y2,col="blue",pch=16)
```

and we can finish by adding a regression line through the extra points:

```
abline(lm(y2~x2))
```

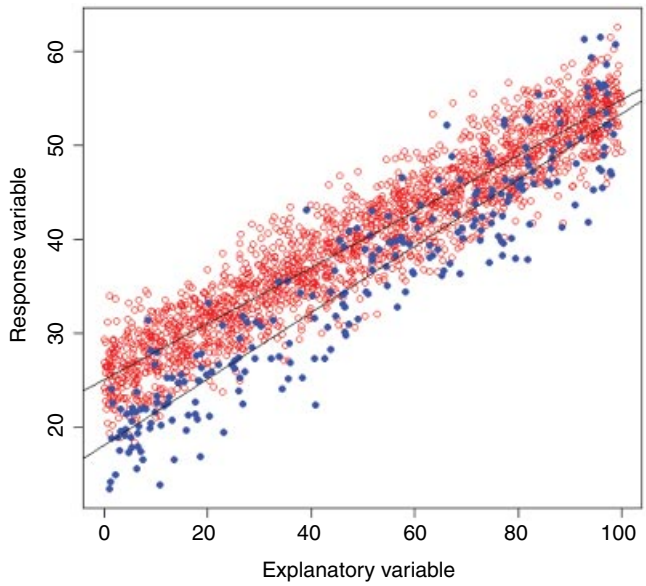


This example shows a very important feature of the `plot` function. Notice that several of the lower values from the second (blue) data set have *not* appeared on the graph. This is because (unless we say otherwise at the outset) R chooses 'pretty' scaling for the axes based on the data range in the *first* set of points to be drawn. If, as here, the range of subsequent data sets lies outside the scale of the *x* and *y* axes, then points are simply left off without any warning message.

One way to cure this problem is to plot *all* the data with `type="n"` so that the axes are scaled to encompass all the points from all the data sets (using the concatenation function, c), then to use `points` and `lines` to add both sets of data to the blank axes, like this:

```
plot(c(x1,x2),c(y1,y2),xlab="Explanatory variable",
                       ylab="Response variable",type="n")
points(x1,y1,col="red")
points(x2,y2,col="blue",pch=16)
```

```
abline(lm(y1~x1))
abline(lm(y2~x2))
```



Now all of the points from both data sets appear on the scatterplot. You might want to take control of the selection of the limits for the *x* and *y* axes, rather than accept the 'pretty' default values. In the last plot, for instance, the minimum on the *y* axis was about 13 (but it is not exactly obvious). You might want to specify that the minimum on the *y* axis was zero. This is achieved with the `ylim` argument, which is a vector of length 2, specifying the minimum and maximum values for the *y* axis: `ylim=c(0,70)`. You will want to control the scaling of the axes when you want two comparable graphs side by side, or when you want to overlay several lines or sets of points on the same axes.

A good way to find out the axis values is to use the `range` function applied to the data sets in aggregate:

```
range(c(x1,x2))
```

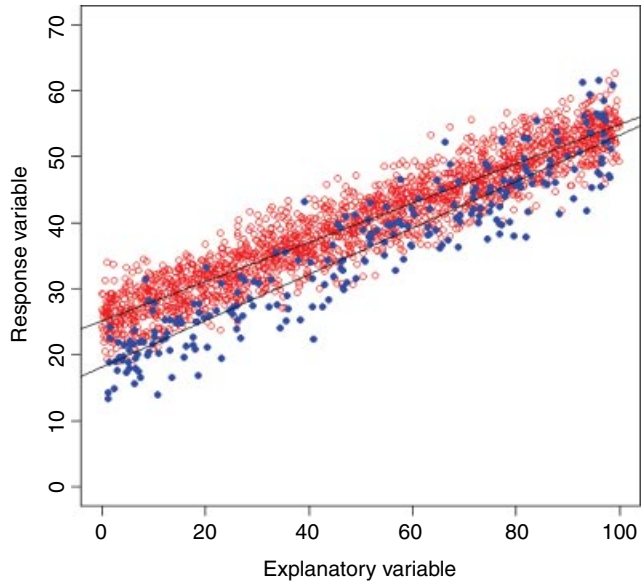
```
[1]  0.02849861 99.93262000
```

```
range(c(y1,y2))
```

```
[1] 13.41794 62.59482
```

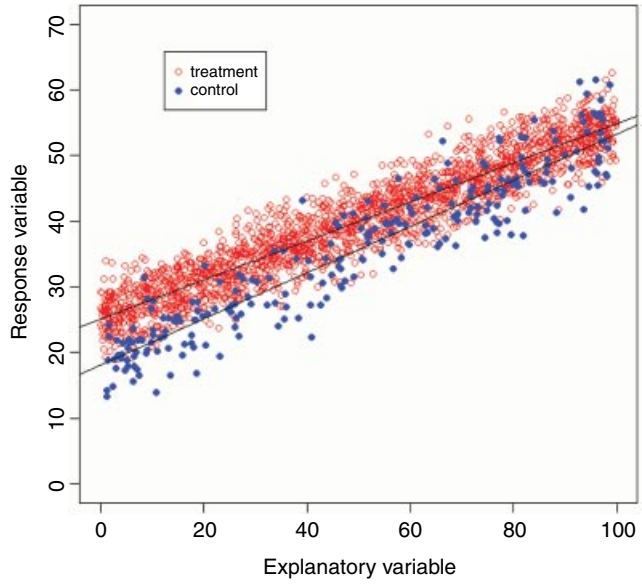
Here the *x* axis needs to go from 0.02 up to 99.93 (0 to 100 would be pretty) and the *y* axis needs to go from 13.4 up to 62.6 (0 to 70 would be pretty). This is how the axes are drawn; the points and lines are added exactly as before:

```
plot(c(x1,x2),c(y1,y2),xlim=c(0,100),ylim=c(0,70),
          xlab="Explanatory variable",ylab="Response variable",type="n")
points(x1,y1,col="red")
points(x2,y2,col="blue",pch=16)
abline(lm(y1~x1))
abline(lm(y2~x2))
```



Adding a legend to the plot to explain the difference between the two colours of points would be useful. The thing to understand about the `legend` function is that the number of lines of text inside the legend box is determined by the length of the vector containing the labels (2 in this case: `c("treatment","control")`. The other two vectors must be of the same length as this: one for the plotting symbols `pch=c(1,16)` and one for the colours `col=c("red","blue")`. The `legend` function can be used with `locator(1)` to allow you to select exactly where on the plot surface the legend box should be placed. Click the mouse button when the cursor is where you want the *top left* of the box around the legend to be.

```
legend(locator(1),c("treatment","control"),pch=c(1,16), col=c("red","blue"))
```

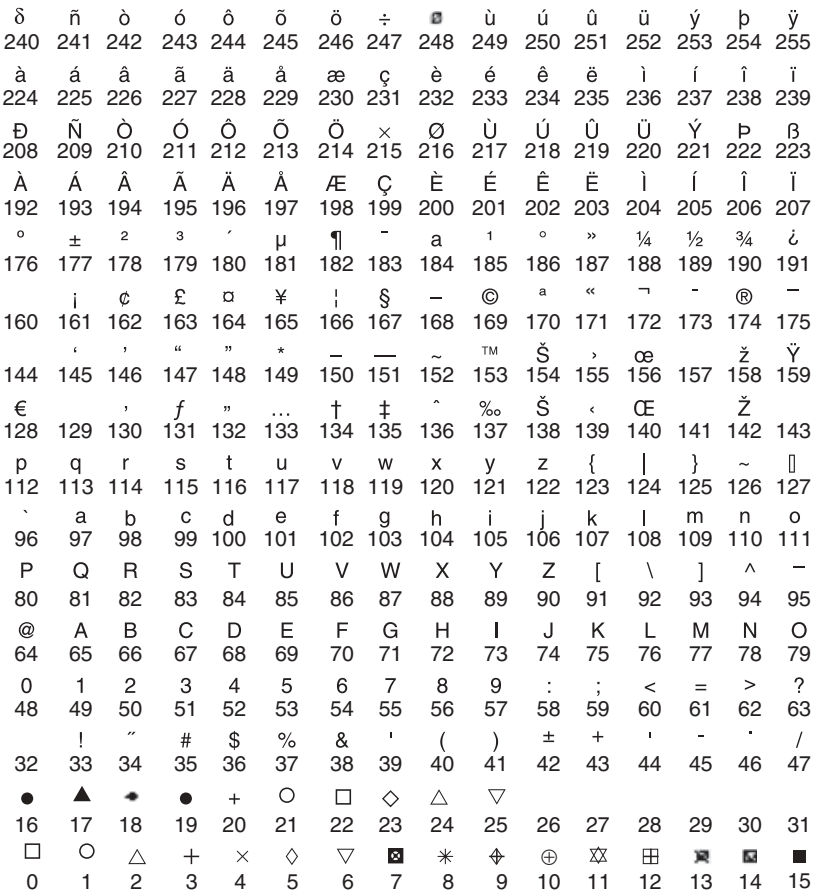


This is about as complicated as you would want to make any figure. Adding more information would begin to detract from the message.

5.2.1 Plotting symbols: `pch`

There are 256 different plotting symbols available in Windows (0 to 255). Here is a graphic showing all of them in sequence, from bottom left to top right:

```
plot(0:10,0:10,xlim=c(0,32),ylim=c(0,40),type="n",xaxt="n",yaxt="n",
xlab="",ylab="")
x <- seq(1,31,2)
s <-  -16
f <-  -1
for (y in seq(2,40,2.5)) {
s <- s+16
f <- f+16
y2 <- rep(y,16)
points(x,y2,pch=s:f,cex=0.7)
text(x,y-1,as.character(s:f),cex=0.6)  }
```

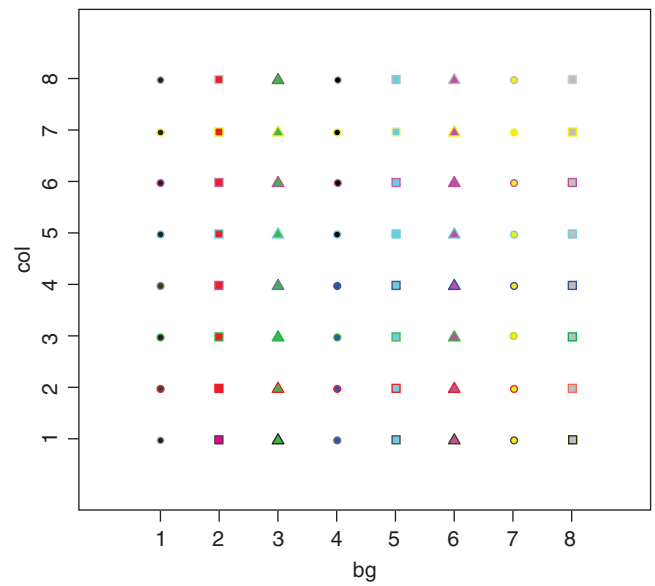


The basic plotting symbols (pch) are shown in the bottom two rows, with their pch number immediately beneath. The default value, pch=1, is a small open circle in black. Note that values between 26 and 32 are not implemented at present and are ignored (blanks are plotted). Values for pch between 33 and 127 represent the ASCII character set, while values between 128 and 255 are the symbols from the Windows character set. The symbols for pch=19 and pch=20 are solid circles of different sizes (pch = 20 is the 'bullet point' and is two-thirds the size of pch=19). The difference between pch=16 and pch=19 is that the latter uses a border, and so it is larger when line width lwd is large relative to character expansion cex. The symbol for pch=46 is the 'dot' and is treated specially (it is a rectangle of side 0.01 inch, scaled by cex, and if cex = 1 (the default), each side is at least one pixel, which is 1/72 inch on the pdf, postscript and xfig devices, so that the dot does not disappear on scaling down the image.

5.2.2 Colour for symbols in plots

The plotting symbols (pch) numbered 21 to 25 allow you to specify the background (fill) colour and the border colour separately. In the illustration below, the background colours (bg) 1 to 8 are shown in the columns and numbered on the *x* axis. The border colours (col) 1 to 8 are shown in the rows and numbered on the *y* axis.

```
plot(0:9,0:9,pch=16,type="n",xaxt="n",yaxt="n",ylab="col",xlab="bg")
axis(1,at=1:8)
axis(2,at=1:8)
for (i in 1:8) points(1:8,rep(i,8),pch=c(21,22,24),bg=1:8,col=i)
```



Some combinations are visually more effective than others. Black borders in row 1 (col = 1) are effective with all the shapes and fill colours, but red borders in row 2 (col = 2) work well only with green, pale blue, yellow and grey fill colours. If you specify only col, then open (white) symbols bordered with the specified colour are produced. If you specify only bg, then solid symbols filled with the specified colour, but bordered in black, are produced. If you specify both col and bg with the same colours, then solid, apparently unbordered symbols are produced. If you specify both col and bg using different colours, then

solid symbols with contrasting borders are produced. Note that for bordered plotting symbols you need to use the argument `pt.bg` in the `legend` function in place of the usual bg for the interior colour of the symbol (because `bg` controls the background colour of the whole legend box).

5.2.3 Adding text to scatterplots

It is very easy to add text to graphics. Suppose you wanted to add the text '(b)' to a plot at the location $x = 80$ and $y = 65$; just type `text(80,65,"(b)")`.

In this example we want to produce a map of place names, and the place names are in a file called `map.places.csv`, but their coordinates are in another, much longer file called `bowens.csv`, containing many more place names than we want to plot. If you have factor level names with spaces in them (e.g. multiple words), then the best format for reading files is comma-delimited (`.csv`) rather than the standard tab-delimited (`.txt`). You read them into a dataframe in R using `read.csv` in place of `read.table`:

```
map.places <- read.csv("c:\\temp\\map.places.csv",header=T)
attach(map.places)
names(map.places)
```

```
[1]  "wanted"
```

```
map.data <- read.csv("c:\\temp\\bowens.csv",header=T)
attach(map.data)
names(map.data)
```

```
[1]  "place"  "east"  "north"
```

There is a slight complication to do with the coordinates. The northernmost places are in a different 100 km square so, for instance, a northing of 3 needs to be altered to 103. It is convenient that all of the values that need to be changed have northings < 60 in the dataframe:

```
nn <- ifelse(north<60,north+100,north)
```

This says change all of the northings for which `north < 60` is `TRUE` to `nn <- north+100`, and leave unaltered all the others (`FALSE`) as `nn <- north`.

The default graphics window in R is a square measuring 7 inches by 7 inches (quaintly old-fashioned, isn't it?). But our map is rectangular, roughly 80 km wide and 50 km high (a rectangle in landscape format). Allowing for the margins, we want to make the plotting region 9 units wide and 7 units high. We achieve this via the `windows` function:

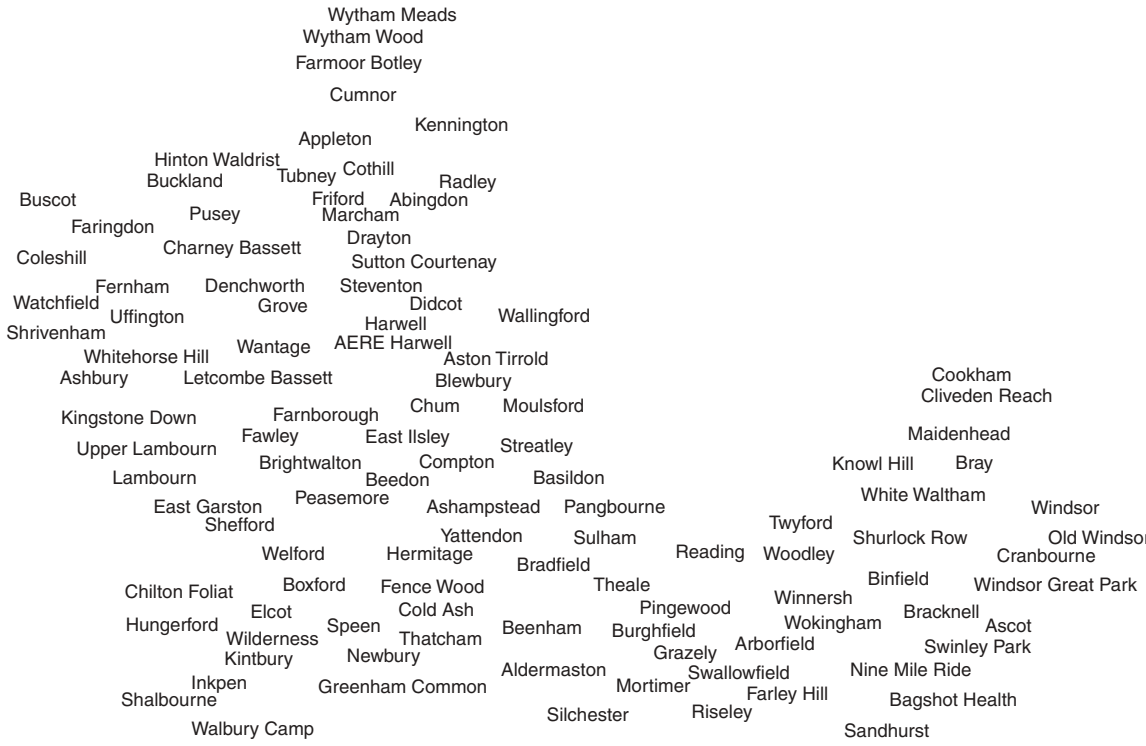
```
windows(9,7)
```

We begin by plotting a blank space (`type="n"`) of the right size (eastings from 20 to 100 and northings from 60 to 110) with blank axis labels and no tick marks or numbers:

```
plot(c(20,100),c(60,110),type="n",xlab="",ylab="",xaxt="n", yaxt="n")
```

The trick is to select the appropriate places in the vector called `place` and use `text` to plot each name in the correct position (`east[ii],nn[ii]`). For each place name in `wanted` we find the correct subscript for that name within `place` using the `which` function to find `ii`:

```
for (i in 1:length(wanted)){
ii <- which(place == as.character(wanted[i]))
text(east[ii], nn[ii], as.character(place[ii]), cex = 0.6) }
```



5.2.4 Identifying individuals in scatterplots

The best way to identify multiple individuals in scatterplots is to use a combination of colours and symbols. A useful trick is to use `as.numeric` to convert a grouping factor (the variable acting as the subject identifier) into a colour and/or a symbol. Here is an example where reaction time is plotted against duration of sleep deprivation for 18 subjects:

```
data <- read.table("c:\\temp\\sleep.txt",header=T)
attach(data)
plot(Days,Reaction)
```

I think you will agree that the raw scatterplot is uninformative; the individuals need to stand out more clearly from one another. The main purpose of the graphic is to show the relationship between sleep deprivation (measured in days) and reaction time. Another aim is to draw attention to the differences between the 18 subjects in their mean reaction times, and to differences in the rate of increase of reaction time with the duration of sleep deprivation. Because there are so many subjects, the graph is potentially very confusing. One improvement is to join together the time series for the individual subjects, using a non-intrusive line colour. Let us do that first. We need to create a vector `s` to contain the numeric values (1 to 18) of the `Subject` identity numbers (which range, with gaps, between 308 and 372):

```
s <- as.numeric(factor(Subject))
```

This vector will be used in subscripts to select the x and y coordinates of each subject's time series in turn. Next, the subjects, `k`, are taken one at a time in a loop, and `lines` with `type="b"` (both points and lines) are drawn in a non-intrusive colour (`gray` is useful for this):

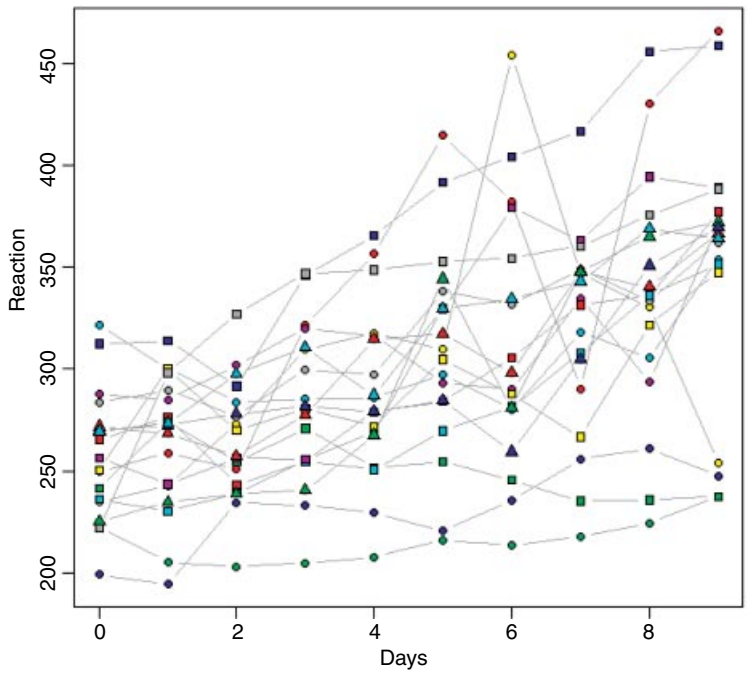
```
plot(Days,Reaction,type="n")
for (k in 1:max(s)){
x <- Days[s==k]
y <- Reaction[s==k]
lines(x,y,type="b",col="gray")
}
```

Next, we need to select plotting symbols and colours for each subject. The colour-filled symbols `pch=21`, `pch=22` and `pch=24` are very useful here. Let us use the non-black colours (`bcol` from 2 to 8) for each of the first two plotting symbols (`sym`), then use colours 2 to 5 for the third plotting symbol for the remaining subjects:

```
sym <- rep(c(21,22,24),c(7,7,4))
bcol <- c(2:8,2:8,2:5)
```

Finally, we can take each subject in turn and use `points` to add the coloured symbols (each with black edges, `col=1`) to the graph:

```
for (k in 1:max(s)){
points(Days[s==k],Reaction[s==k],pch=sym[k],bg=bcol[k],col=1)
}
```



I think that there is insufficient room on the plotting surface to insert a legend with 18 labels in it. For a plot as complicated as this, it is best to put the explanations of the plotting symbols in the text. Perhaps the clearest pattern to emerge from the graphic is that subject 331 (the yellow-filled circle) clearly had a hangover on day 6, because he/she was the third fastest reactor after 9 days of deprivation.

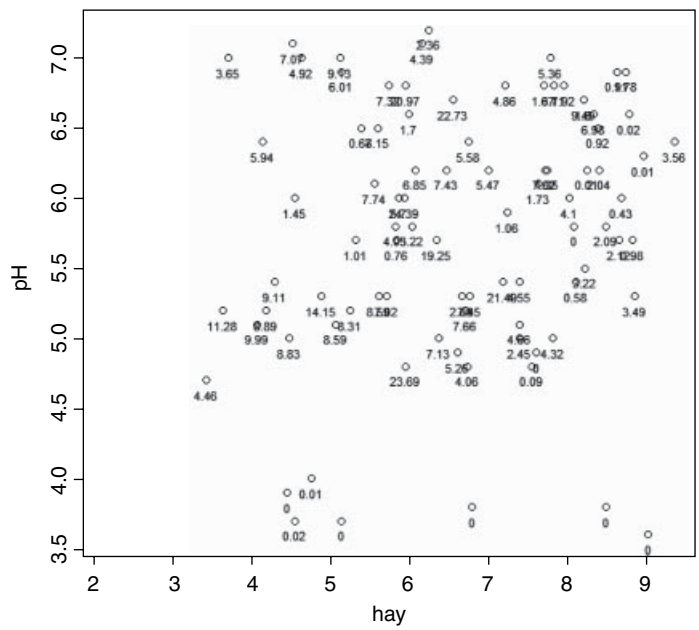
5.2.5 Using a third variable to label a scatterplot

The following example concerns the response of a grass species *Festuca rubra* as measured by its biomass in small samples (FR) to two explanatory variables, soil pH and total hay yield (the mass of all plant species combined). A scatterplot of pH against hay shows the locations of the various samples. The idea is to use the text function to label each of the points on the scatterplot with the dry mass of *F. rubra* in that particular sample, to see whether there is systematic variation in the mass of *Festuca* with changes in hay yield and soil pH.

```
data <- read.table("c:\\temp\\pgr.txt",header=T)
attach(data)
names(data)
```

```
[1]  "FR"  "hay"  "pH"
```

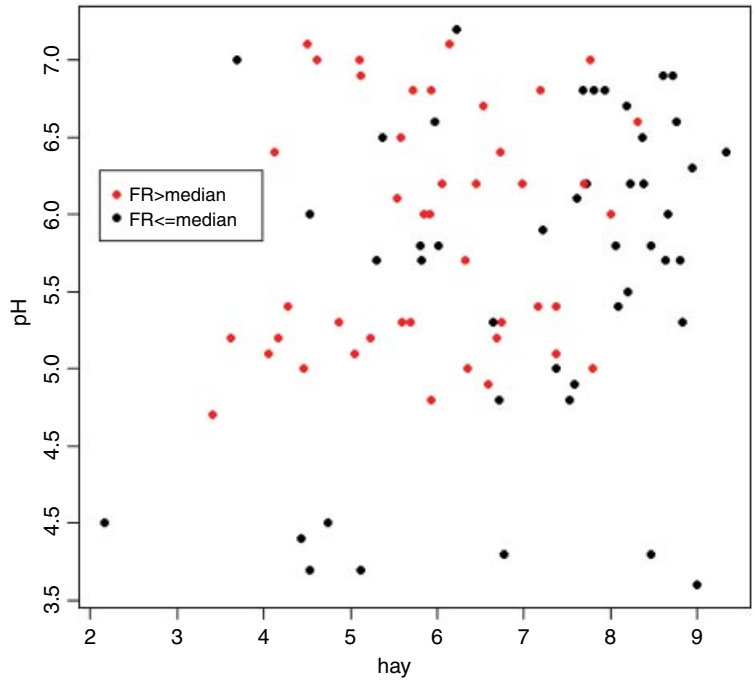
```
plot(hay,pH)
text(hay, pH, labels=round(FR, 2), pos=1, offset=0.5,cex=0.7)
```



The labels are *centred* on the *x* value of the point (pos=1) and are *offset half a character below* the point (offset=0.5). They show the value of FR rounded to two significant digits (labels=round(FR, 2)) at 70% character expansion (cex=0.7). There is an obvious problem with this method when there is lots of overlap between the labels (as in the top right), but the technique works well for more widely spaced points. The plot shows that high values of *Festuca* biomass are concentrated at intermediate values of both soil pH and hay yield.

You can also use a third variable to choose the colour of the points in your scatterplot. Here the points with FR above median are shown in red, the others in black:

```
plot(hay,pH,pch=16,col=ifelse(FR>median(FR),"red","black"))
legend(locator(1),c("FR>median","FR<=median"),pch=16,col=c("red","black"))
```



For three-dimensional plots see `image`, `contour` and `wireframe` on p. 931.

5.2.6 Joining the dots

Sometimes you want to join the points on a scatterplot by lines. The trick is to ensure that the points on the *x* axis are ordered: if they are not ordered, the result is a mess, as you will see below.

```
smooth <- read.table("c:\\temp\\smoothing.txt",header=T)
attach(smooth)
names(smooth)
```

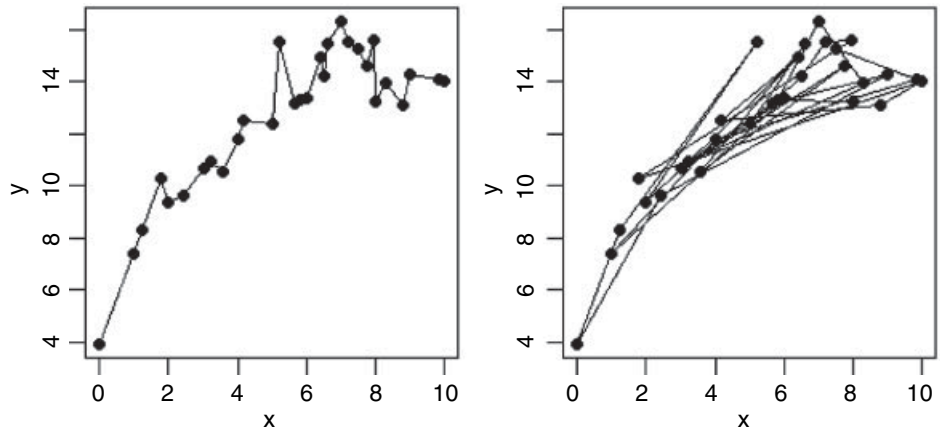
```
[1]  "x"  "y"
```

Begin by producing a vector of subscripts representing the ordered values of the explanatory variable. Then draw lines with this vector as subscripts to both the *x* and *y* variables:

```
plot(x,y,pch=16)
sequence <- order(x)
lines(x[sequence],y[sequence])
```

If you do not order the *x* values, and just use the `lines` function, this is what happens:

```
plot(x,y,pch=16)
lines(x,y)
```



There is a plot option `type="b"` (this stands for 'both' points and lines) which draws the points and joins them together with lines. You can choose the plotting symbol (`pch`) and the line type (`lty`) to be used.

5.2.7 Plotting stepped lines

When plotting square edges between two points, you need to decide whether to go across and then up, or up and then across. The issue should become clear with an example. We have two vectors from 0 to 10:

```
x<-0:10
y<-0:10
plot(x,y)
```

There are three ways we can join the dots: with a straight line

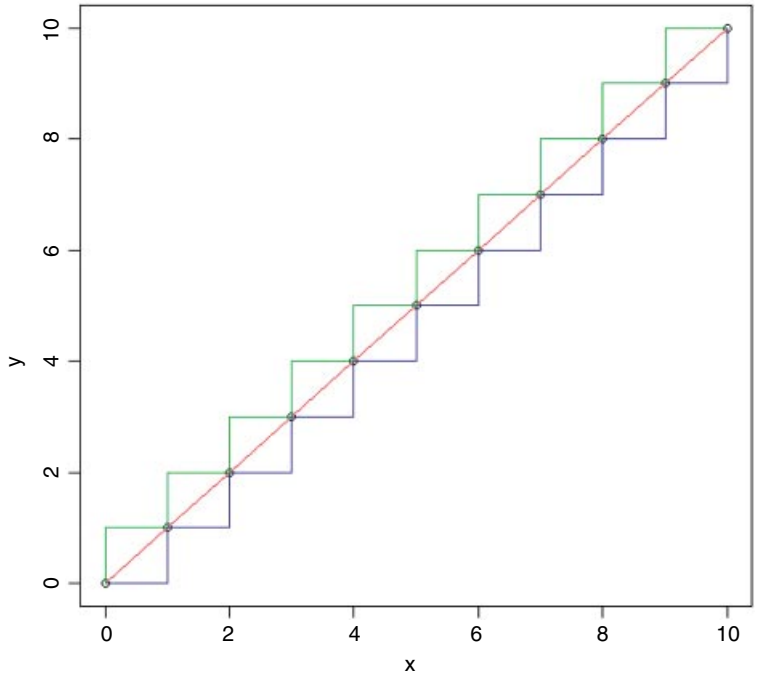
```
lines(x,y,col="red")
```

with a stepped line going across first then up, using lower-case 's'

```
lines(x,y,col="blue",type="s")
```

or with a stepped green line going up first, then across using upper-case 'S' ('upper case, up first' is the way to remember it):

```
lines(x,y,col="green",type="S")
```



5.3 Adding other shapes to a plot

Once you have produced a set of axes using `plot` it is straightforward to locate and insert other kinds of things. Here are two unlabelled axes, without tick marks (`xaxt="n"`), both scaled from 0 to 10 but without any of the 11 points drawn on the axes (`type="n"`):

```
plot(0:10,0:10,xlab="",ylab="",xaxt="n",yaxt="n",type="n")
```

You can easily add extra graphical objects to plots:

- `rect` rectangles
- `arrows` arrows and headed bars
- `polygon` more complicated filled shapes, including objects with curved sides

For the purposes of demonstration we shall add a single-headed arrow, a double-headed arrow, a rectangle and a six-sided polygon to this space.

We want to put a solid square object in the top right-hand corner, and we know the precise coordinates to use. The syntax for the `rect` function is to provide four numbers:

```
rect(xleft, ybottom, xright, ytop)
```

Thus, to plot the square from (6,6) to (9,9) involves:

```
rect(6,6,9,9)
```

You can fill the shape with solid colour (col) or with shading lines (density, angle) as described on p. 920.

5.3.1 Placing items on a plot with the cursor, using the locator function

You might want to point with the cursor and get R to tell you the coordinates of the corners of the rectangle. You can use the locator() function for this. The rect function does not accept locator as its argument, but you can easily write a function (here called corners) to do this:

```
corners <- function(){
      coos <- c(unlist(locator(1)),unlist(locator(1)))
      rect(coos[1],coos[2],coos[3],coos[4])
}
```

Run the function like this:

```
corners()
```

Then click in the bottom left-hand corner and again in the top right-hand corner, and a rectangle will be drawn from your screen-supplied pointers.

Drawing arrows is straightforward. The syntax for the arrows function is to draw a line from the point (x0, y0) to the point (x1, y1) with the arrowhead, by default, at the 'second' end (x1, y1):

```
arrows(x0, y0, x1, y1)
```

Thus, to draw an arrow from (1,1) to (3,8) with the head at (3,8) type:

```
arrows(1,1,3,8)
```

A horizontal double-headed arrow from (1,9) to (5,9) is produced by adding code=3 like this:

```
arrows(1,9,5,9,code=3)
```

A vertical bar with two square ends (e.g. like an error bar) uses angle = 90 instead of the default angle = 30):

```
arrows(4,1,4,6,code=3,angle=90)
```

Here is a function that draws an arrow from the cursor position of your first click to the position of your second click:

```
click.arrows <- function(){
      coos <- c(unlist(locator(1)),unlist(locator(1)))
      arrows(coos[1],coos[2],coos[3],coos[4])
}
```

To run this, type

```
click.arrows()
```

then click the cursor on the two ends.

We now wish to draw a polygon. To do this, it is often useful to save the values of a series of locations. Here we intend to save the coordinates of six points in a vector called locations to define a polygon for plotting:

```
locations <- locator(6)
```

After you have clicked over the sixth location, control returns to the screen. What kind of object has locator produced?

```
class(locations)
```

```
[1]  "list"
```

It has produced a `list`, and we can extract the vectors of x and y values from the list using `$` to name the elements of the list (R has created the helpful names `x` and `y`):

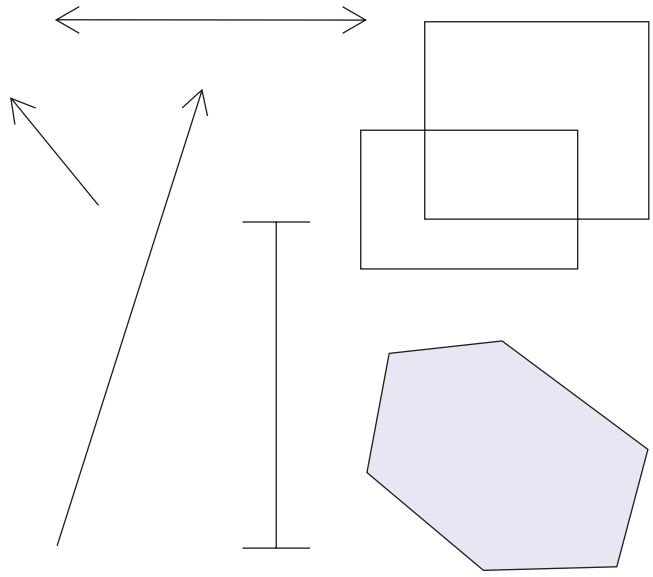
```
locations
```

```
$x
[1] 5.484375 7.027344 9.019531 8.589844 6.792969 5.230469

$y
[1] 3.9928797 4.1894975 2.5510155 0.7377620 0.6940691 2.1796262
```

Now we draw a lavender-coloured polygon like this:

```
polygon(locations,col="lavender")
```



Note that the `polygon` function has automatically closed the shape, drawing a line from the last point to the first.

5.3.2 Drawing more complex shapes with `polygon`

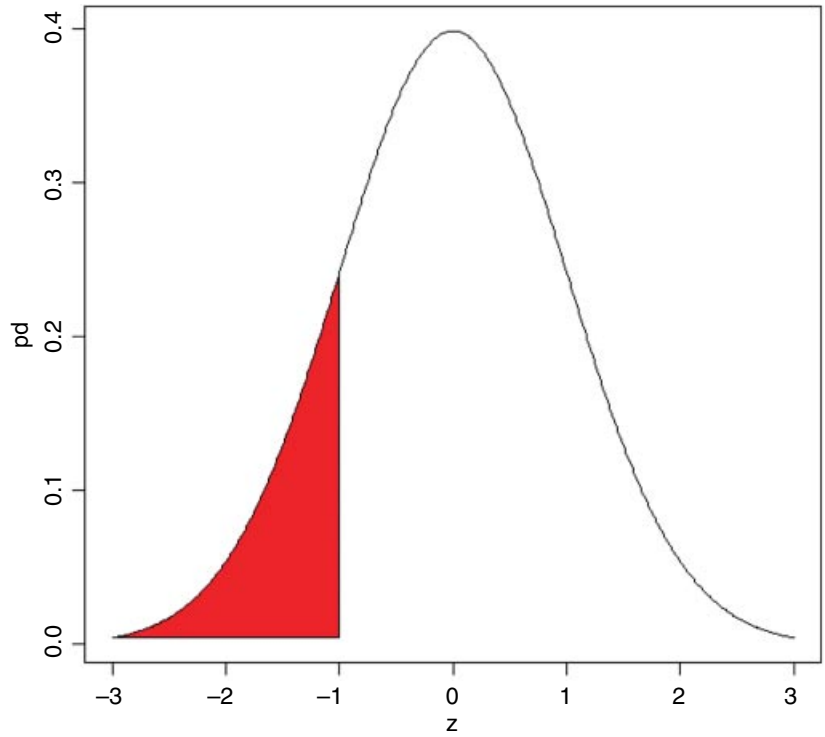
The `polygon` function can be used to draw more complicated shapes, including curved ones. In this example we are asked to shade the area beneath a standard normal curve for values of `z` that are less than or equal

to –1. First draw the probability density (dnorm) line for the standard normal (mean = 0 and standard deviation = 1):

```
z <- seq(-3,3,0.01)
pd <- dnorm(z)
plot(z,pd,type="l")
```

Then fill the area to the left of z ≤ –1 in red:

```
polygon(c(z[z<=-1],-1),c(pd[z<=-1],pd[z==-3]),col="red")
```

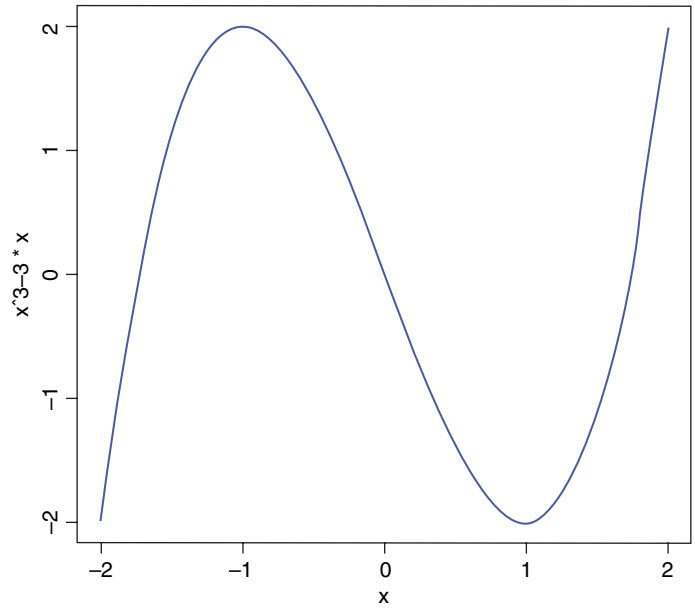


Note the insertion of the point (-1, pd[z == -3]) to create the right-angled corner to the polygon on the z axis at z = –1 and pd set to the same value as when z is –3 to make sure that the bottom line is horizontal.

5.4 Drawing mathematical functions

The curve function is convenient for this. Here is a plot of $x^3 - 3x$ between $x = -2$ and $x = 2$:

```
curve(x^3-3*x, -2, 2)
```



Here is the more cumbersome code to do the same thing using `plot`:

```
x <- seq(-2,2,0.01)
y <- x^3-3*x
plot(x,y,type="l")
```

With `plot`, you need to decide how many segments you want to generate to create the curve (using `seq` with steps of 0.01 in this example), then calculate the matching `y` values, then use `plot` with `type="l"`. This stands for 'type = line' (rather than the default `points`) and can cause problems if you misread the symbol as a number 'one' rather than a lower-case letter 'L'.

5.4.1 Adding smooth parametric curves to a scatterplot

Up to this point our response variable was shown as a scatter of data points. In many cases, however, we want to show the response as a smooth curve. The important tip is that to produce reasonably smooth-looking curves in R you should draw about 100 straight-line sections between the minimum and maximum values of your x axis.

The Ricker curve is named after the famous Canadian fish biologist who introduced this two-parameter hump-shaped model for describing recruitment to a fishery y as a function of the density of the parental stock, x. We wish to compare two Ricker curves with the following parameter values:

$$y_A = 482x\,e^{-0.045x}, \qquad y_B = 518x\,e^{-0.055x}.$$

The first decision to be made is the range of x values for the plot. In our case this is easy because we know from the literature that the minimum value of x is 0 and the maximum value of x is 100. Next we need to generate about 100 values of x at which to calculate and plot the smoothed values of y:

```
xv <- 0:100
```

Next, calculate vectors containing the values of y_A and y_B at each of these x values:

```
yA <- 482*xv*exp(-0.045*xv)
yB <- 518*xv*exp(-0.055*xv)
```

We are now ready to draw the two curves, but we do not know how to scale the y axis. We could find the maximum and minimum values of y_A and y_B then use `ylim` to specify the extremes of the y axis, but it is more convenient to use the option `type="n"` to draw the axes without any data, then use `lines` to add the two smooth functions later. The blank axes are produced like this:

```
plot(c(xv,xv),c(yA,yB),xlab="stock",ylab="recruits",type="n")
```

We want to draw the smooth curve for y_A as a dashed blue line (`lty = 2, col = "blue"`),

```
lines(xv,yA,lty=2,col="blue")
```

and the curve for y_B as a solid red line (`lty = 1, col = "red"`),

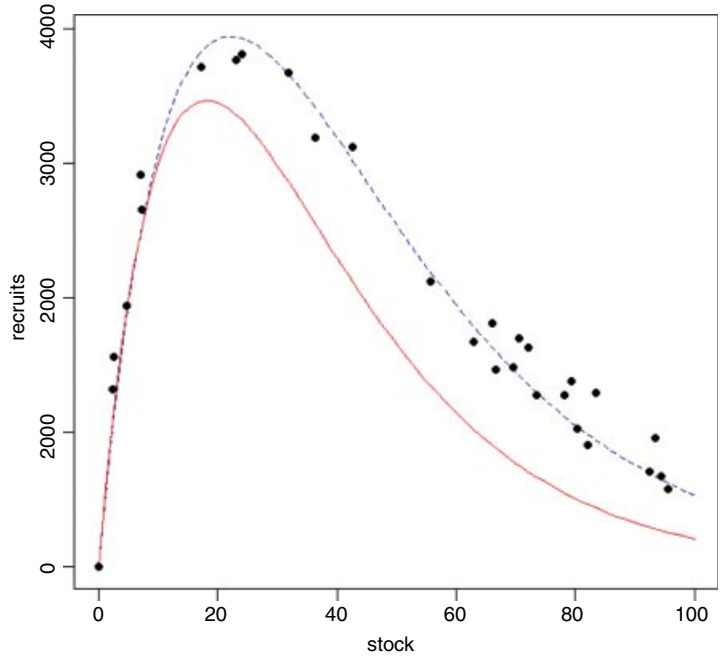
```
lines(xv,yB,lty=1,col="red")
```

Next, we want to see which (if either) of these lines best describes our field data, by overlaying a scatter of points (as black solid circles, `pch = 16`) on the smooth curves:

```
info <- read.table("c:\\temp\\plotfit.txt",header=T)
attach(info)
names(info)
```

```
[1] "x" "y"
```

```
points(x,y,pch=16)
```



You can see that the blue dotted line is a much better description of our data than is the solid red line. Estimating the parameters of non-linear functions like the Ricker curve from data is explained in Chapter 20.

5.4.2 Fitting non-parametric curves through a scatterplot

It is common to want to fit a non-parametric smoothed curve through data, especially when there is no obvious candidate for a parametric function. R offers a range of options:

- `lowess` (a non-parametric curve fitter);
- `loess` (a modelling tool);
- `gam` (fits generalized additive models; p. 666);
- `lm` for polynomial regression (fit a linear model involving powers of x).

We will illustrate each of these options using the jaws data. First, we load the data:

```
data <- read.table("c:\\temp\\jaws.txt",header=T)
attach(data)
names(data)
```

```
[1]  "age"  "bone"
```

Before we fit our various curves to the data, we need to consider how best to display the results together. Without doubt, the graphical parameter you will change most often just happens to be the least intuitive to use. This is the number of graphs per screen, called somewhat unhelpfully, `mfrow`. This stands for 'multiple frames by rows'. The idea is simple, but the syntax is hard to remember. You need to specify the number of rows of plots you want, and number of plots per row, in a vector of two numbers. The first number is the number of rows and the second number is the number of graphs per row. The vector is made using concatenate `c` in the normal way. The default single-plot screen is `par(mfrow=c(1,1))`. Two plots side by side is `par(mfrow=c(1,2))` and a panel of four plots in a 2×2 square is `par(mfrow=c(2,2))`.

To move from one plot to the next, you need to execute a new `plot` function. Control stays within the same plot frame while you execute functions like `points`, `lines` or `text`. Remember to return to the default single plot when you have finished your multiple plot by executing `par(mfrow=c(1,1))`. If you have more than two graphs per row or per column, the character expansion `cex` is set to 0.5 and you get half-size characters and labels.

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

Let us now plot our four graphs with different smooth functions fitted through the jaws data. First, the simple non-parametric smoother called `lowess`. You provide the `lowess` function with arguments for the explanatory variable and the response variable, then provide this object as an argument to the `lines` function like this:

```
plot(age,bone,pch=16,main="lowess")
lines(lowess(age,bone),col="red")
```

It is a reasonable fit overall, but a poor descriptor of the jaw size for the lowest five ages. Let us try `loess`, which is a model-fitting function. We use the fitted model to `predict` the jaw sizes:

```
plot(age,bone,pch=16,main="loess")
model <- loess(bone~age)
xv <- 0:50
yv <- predict(model,data.frame(age=xv))
lines(xv,yv,col="red")
```

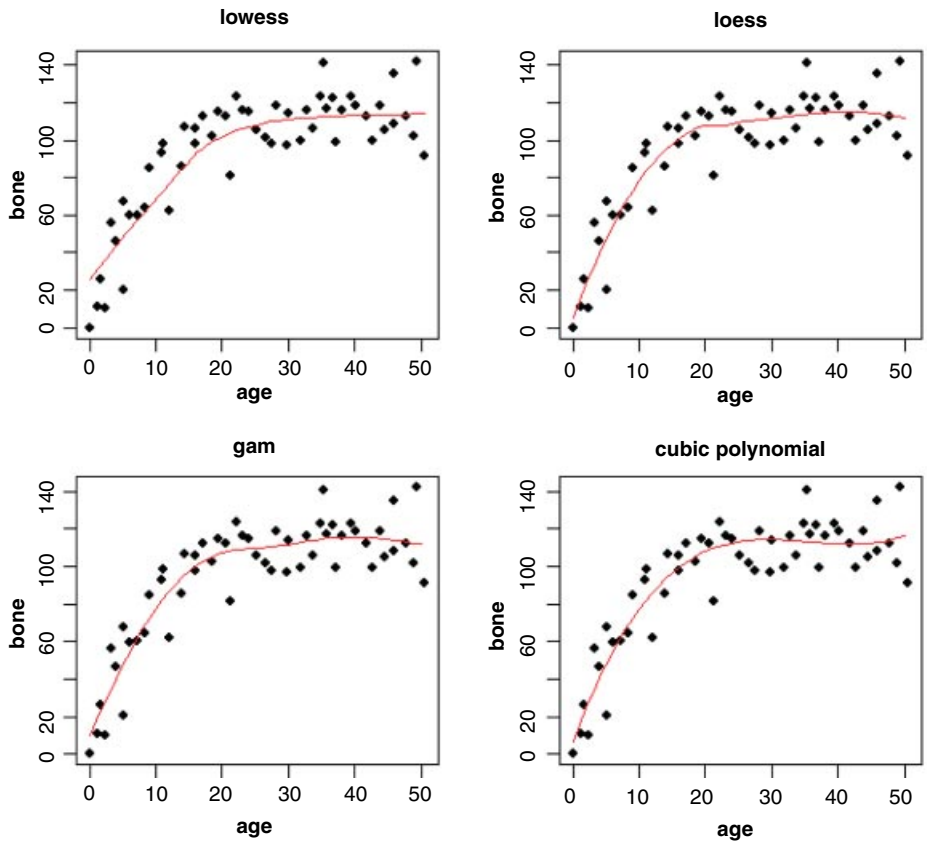
This is much better at describing the jaw size of the youngest animals, but shows a slight decrease for the oldest animals which might not be realistic. Next, we use a generalized additive model (`gam`, from the library `mgcv`) to fit bone as `s(age),` a smooth function of age:

```
library(mgcv)
plot(age,bone,pch=16,main="gam")
model <- gam(bone~s(age))
xv <- 0:50
yv <- predict(model,list(age=xv))
lines(xv,yv,col="red")
```

The line is almost indistinguishable from the line produced by `loess`. Finally, a polynomial:

```
plot(age,bone,pch=16,main="cubic polynomial"))
model <- lm(bone~age+I(age^2)+I(age^3))
xv <- 0:50
yv <- predict(model,list(age=xv))
lines(xv,yv,col="red")
```

As so often with polynomials, the line is more curvaceous than we really want. Note the use of capital `I` (the 'as is' function) in front of the quadratic and cubic terms. The fit is good for young animals, but is rather wavy where we might expect to see an asymptote. It tips up at the end, whereas the last two smoothers tipped down.



Because it is a built-in function and does not require any external packages to be loaded, my recommendation is for `loess` (top right); it is a reasonable fit, and is not over-curvaceous. You fit a model, then use `predict` with a specified vector of values for the explanatory variable, then draw the curve using `lines`.

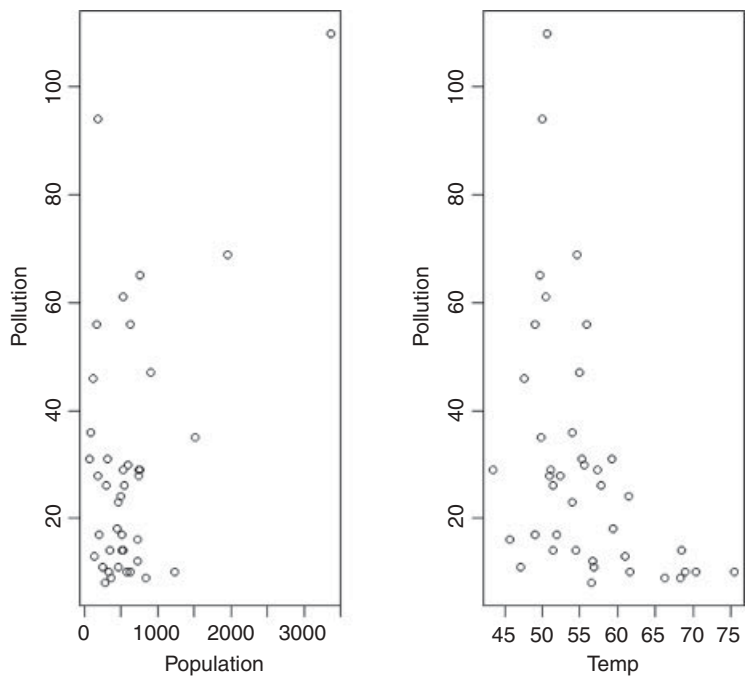
5.5 Shape and size of the graphics window

The default graphics window is a square, measuring 7 inches by 7 inches (I know it should be metric, but it is not). This is fine for most purposes, but it needs to be changed if you want to put two graphs side by side, using `par(mfrow=c(1,2))`.

```
data <- read.table("c:\\temp\\pollute.txt",header=T)
attach(data)
```

If you use the default window the graphs will come out looking far too narrow:

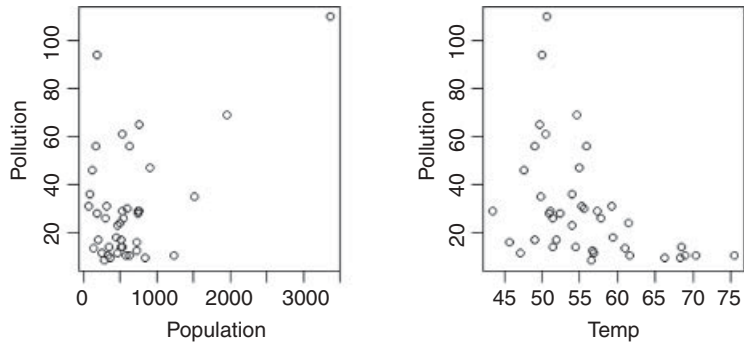
```
par(mfrow=c(1,2))
plot(Population,Pollution)
plot(Temp,Pollution)
```



The simplest solution is to use the mouse to drag up the base of the graphics window until you obtain a more pleasing shape. Alternatively, you can invoke the `windows` function, specifying first the width then

the height in inches. The best choice for this case is (7,4):

```
windows(7,4)
par(mfrow=c(1,2))
plot(Population,Pollution)
plot(Temp,Pollution)
```



5.6 Plotting with a categorical explanatory variable

When the explanatory variable is categorical rather than continuous, we cannot produce a scatterplot. Instead, we choose between a **barplot** and a **boxplot**. I prefer box-and-whisker plots because they convey so much more information, and this is the default plot in R with a categorical explanatory variable.

Categorical variables are **factors** with two or more **levels** (see p. 20). Our first example uses the factor called `month` (with levels 1 to 12) to investigate weather patterns at Silwood Park:

```
weather <- read.table("c:\\temp\\SilwoodWeather.txt",header=T)
attach(weather)
names(weather)
```

```
[1]  "upper"  "lower"  "rain"  "month"  "yr"
```

There is one bit of housekeeping we need to do before we can plot the data. We need to declare `month` to be a factor. At the moment, R thinks it is just a number:

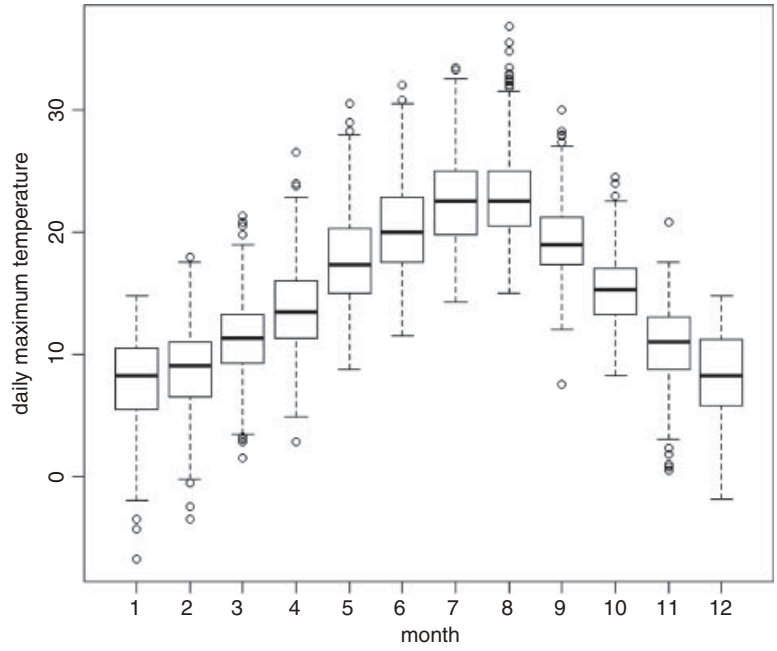
```
month <- factor(month)
```

Now we can plot using a categorical explanatory variable (`month`) and, because the first variable is a factor, we get a boxplot rather than a scatterplot:

```
plot(month,upper)
```

Note that there are no axis labels in the default box-and-whisker plot, and to get informative labels we should need to type:

```
plot(month,upper,ylab="daily maximum temperature",xlab="month")
```



The boxplot summarizes a great deal of information very clearly. The horizontal line shows the **median** upper daily temperature for each month. The bottom and top of the box show the 25th and 75th **percentiles**, respectively (i.e. the location of the middle 50% of the data, also called the first and third **quartiles**). The vertical dashed lines are called the 'whiskers'. For the upper whisker, we see one of two things: either the maximum value or, when there are outliers present, the largest data point that is less than 1.5 times the **interquartile range** above the 75th percentile. The quantity '1.5 times the interquartile range of the data' is roughly 2 standard deviations, and the interquartile range is the difference in the response variable between its first and third quartiles. Points more than 1.5 times the interquartile range *above the third quartile* and points more than 1.5 times the interquartile range *below the first quartile* are defined as **outliers** and plotted individually. Thus, when there are no outliers the whiskers simply show the maximum and minimum values (as shown here only in month 12). Boxplots not only show the location and spread of data but also indicate skewness (which shows up as asymmetry in the sizes of the upper and lower parts of the box). For example, in February the range of lower temperatures was much greater than the range of higher temperatures. Boxplots are also excellent for spotting errors in the data when the errors are represented by extreme outliers. Note that the box-and-whisker plot is based entirely on the data points themselves; there are no estimated parameters like means or standard deviations. The whiskers always end at data points, so the upper and lower whiskers are typically asymmetric, even when there are outliers both above and below (e.g. in November).

5.6.1 Boxplots with notches to indicate significant differences

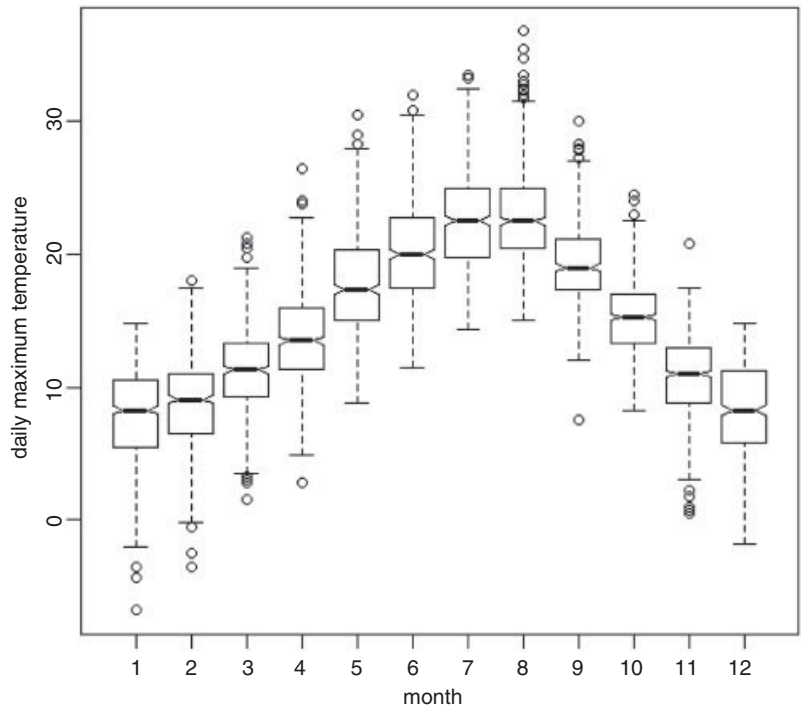
Boxplots are very good at showing the distribution of the data points around the median, but they are not so good at indicating whether or not the median values are significantly different from one another. Tukey invented **notches** to get the best of both worlds. The notches are drawn as a 'waist' on either side of the median and are intended to give a rough impression of the significance of the differences between two medians. Boxes in which *the notches do not overlap* are likely to prove to have significantly different medians under an appropriate test. Boxes with overlapping notches probably do not have significantly different medians. The

size of the notch increases with the magnitude of the interquartile range and declines with the square root of the replication, like this:

$$\text{notch} = \pm 1.58 \frac{\text{IQR}}{\sqrt{n}},$$

where IQR is the interquartile range and n is the replication per sample. Notches are based on assumptions of asymptotic normality of the median and roughly equal sample sizes for the two medians being compared, and are said to be rather insensitive to the underlying distributions of the samples. The idea is to give roughly a 95% confidence interval for the difference in two medians, but the theory behind this is somewhat vague.

Here are the Silwood Weather data (above) with the option `notches=TRUE`:



There is no significant difference in daily maximum temperature between July and August (the notches for months 7 and 8 overlap completely), but maxima in September are significantly lower than in August. If the boxes do not overlap (e.g. months 9 and 10) then the difference in their medians will be highly significant under the appropriate test.

When the sample sizes are small and/or the within-sample variance is high, the notches are not drawn as you might expect them (i.e. as a waist within the box). Instead, the notches are extended *above* the 75th percentile and/or *below* the 25th percentile. This looks odd, but it is an intentional feature, supposed to act as a warning of the likely invalidity of the test (see p. 217).

5.6.2 Barplots with error bars

Rather than use `plot` to produce a boxplot, an alternative is to use a `barplot` to show the heights of the mean values from the different treatments. We need to begin by calculating the heights of the bars, typically by using the function `tapply` to work out the mean values for each level of the categorical explanatory

variable. Data for this example come from an experiment on plant competition, with five factor levels in a single categorical variable called `clipping`: a control (unclipped), two root clipping treatments (`r5` and `r10`) and two shoot clipping treatments (`n25` and `n50`) in which the leaves of neighbouring plants were reduced by 25% and 50%. The response variable is yield at maturity (a dry weight) called `biomass`.

```
trial <- read.table("c:\\temp\\compexpt.txt",header=T)
attach(trial)
names(trial)
```

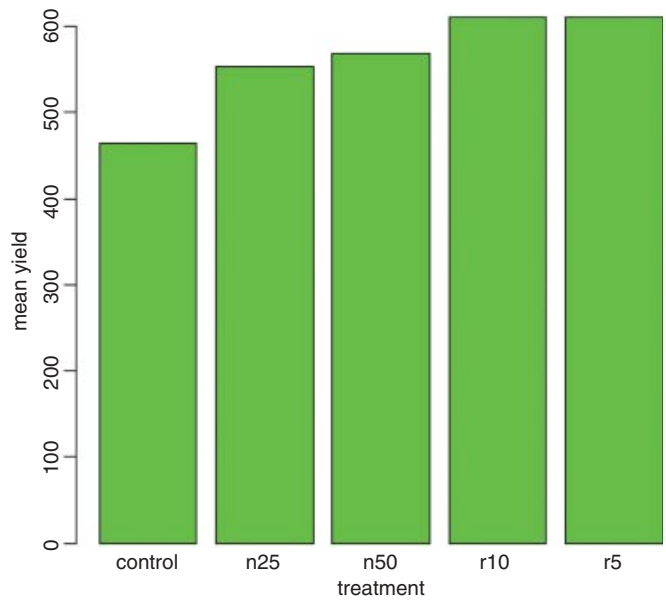
```
[1] "biomass"  "clipping"
```

First, calculate the heights of the bars using `tapply` to compute the five mean values:

```
means <- tapply(biomass,clipping,mean)
```

Then the barplot is produced very simply:

```
barplot(means,xlab="treatment",ylab="mean yield",col="green")
```



Unless we add error bars to such a barplot, the graphic gives no indication of the extent of the uncertainty associated with each of the estimated treatment means, and hence is unsuitable for publication. There is no built-in function for drawing error bars on barplots, but it easy to write a function to do this. One obvious issue is that the *y* axis as drawn by the previous call to `barplot` is likely to be too short to accommodate the error bar extending from the top of the tallest bar. Another issue is that it is not obvious where to centre each of the error bars (i.e. the *x* coordinates of the middles of the bars).

The next decision to make is what kind of bar to draw. Many journals prefer plus or minus one standard error of the mean. An old fashioned approach is to use plus or minus the 95% confidence interval of the mean. Perhaps the most informative error bar is plus or minus one half of the least significant difference between two means (because then non-overlapping bars indicates significant difference, and overlapping bars indicates non-significance; see p. 515). On the assumption that you want to publish your work in *Science* or *Nature*, we shall use plus or minus one standard error of the mean, because this is their error bar of choice. First, work out the error variance from the ANOVA table of `lm(y~x)` where `x` is categorical. Now calculate

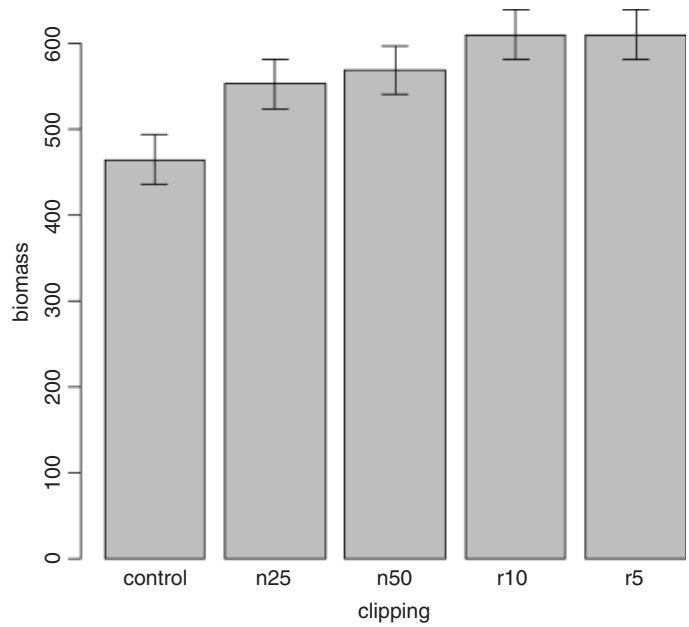
the replication per factor level, and use this to compute `sem`, the standard error of the mean. Work out the mean values that will be represented by the heights of the bars using `tapply`. To scale the top of the *y* axis, add the standard error to the largest of the means. Determine the labels for the bars from the factor levels of the explanatory variable using `nn <- as.character(levels(x))`. Find the locations of the centres of the bars along the *x* axis using `xs <- barplot`. Here is the function in full:

```
seBars <- function(x,y){
      model <- lm(y~factor(x))
      reps <- length(y)/length(levels(x))
      sem <- summary(model)$sigma/sqrt(reps)
      m <- as.vector(tapply(y,x,mean))
      upper <- max(m)+sem
      nn <- as.character(levels(x))
      xs <- barplot(m,ylim=c(0,upper),names=nn,
            ylab=deparse(substitute(y)),xlab=deparse(substitute(x)))

for (i in 1:length(xs)) {
      arrows(xs[i],m[i]+sem,xs[i],m[i]-sem,angle=90,code=3,length=0.1) }
}
```

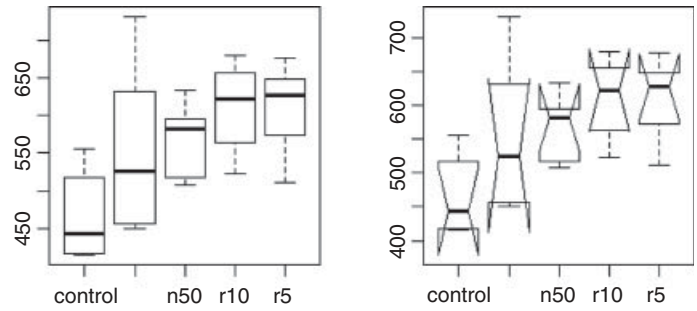
You run it like this, specifying the categorical variable first, then the continuous response variable:

```
seBars(clipping,biomass)
```



For comparison, here are the box-and-whisker plots for the same data, without and with notches:

```
windows(7,4)
par(mfrow=c(1,2))
plot(clipping,biomass)
plot(clipping,biomass,notch=T)
```



illustrating the curious behaviour of the notches when the sample sizes are small.

5.6.3 Plots for multiple comparisons

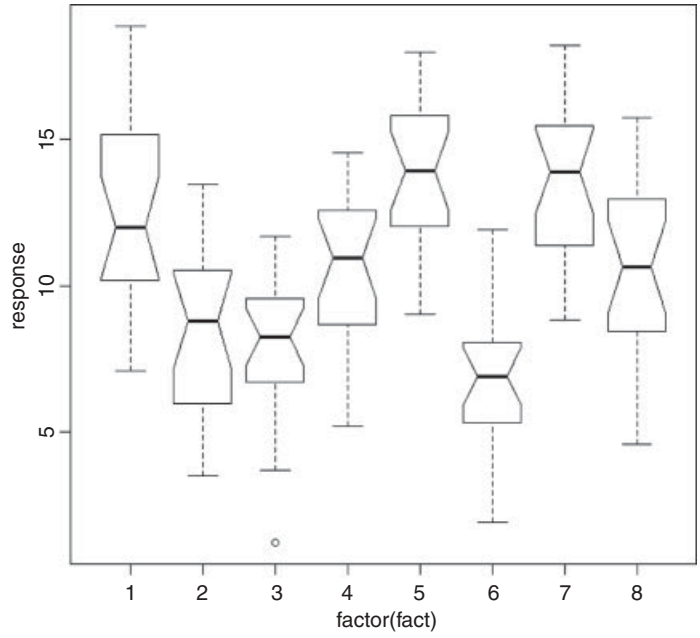
When there are many levels of a categorical explanatory variable, we need to be cautious about the statistical issues involved with multiple comparisons (see p. 531). Here we contrast two graphical techniques for displaying multiple comparisons: boxplots with notches, and Tukey's 'honest significant difference'.

The data show the response of yield to a categorical variable (`fact`) with eight levels representing eight different genotypes of seed (cultivars) used in the trial:

```
data <- read.table("c:\\temp\\box.txt",header=T)
attach(data)
names(data)
```

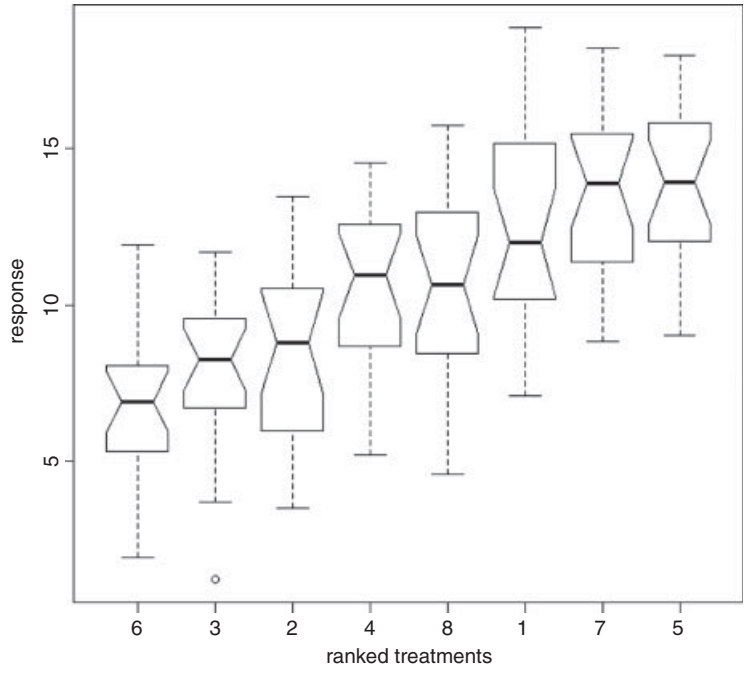
```
[1] "fact"  "response"
```

```
plot(response~factor(fact),notch=TRUE)
```



Because the genotypes (factor levels) are unordered, it is hard to judge from the plot which levels might be significantly different from which others. We start, therefore, by calculating an index which will rank the mean values of response across the different factor levels:

```
index <- order(tapply(response,fact,mean))
ordered <- factor(rep(index,rep(20,8)))
boxplot(response~ordered,notch=T,names=as.character(index),
  xlab="ranked treatments",ylab="response")
```



There are several points to clarify here. We plot the response as a function of the factor called `ordered` (rather than `fact`) so that the boxes are ranked from lowest mean yield on the left (cultivar 6) to greatest mean on the right (cultivar 5). We change the names of the boxes to reflect the values of `index` (i.e. the original values of `fact`: otherwise they would read 1 to 8). Note that the vector called `index` is of length 8 (the number of boxes on the plot), but `ordered` is of length 160 (the number of values of response). Looking at the notches, no two adjacent pairs of medians appear to be significantly different, but the median of treatment 4 appears to be significantly greater than the median of treatment 6, and the median of treatment 5 appears to be significantly greater than the median of treatment 8 (but only just).

The statistical analysis of these data might involve user-specified contrasts (p. 434), once it is established that there are significant differences to be explained. This we assess with a one-way analysis of variance to test the hypothesis that at least one of the means is significantly different from the others (see p. 501):

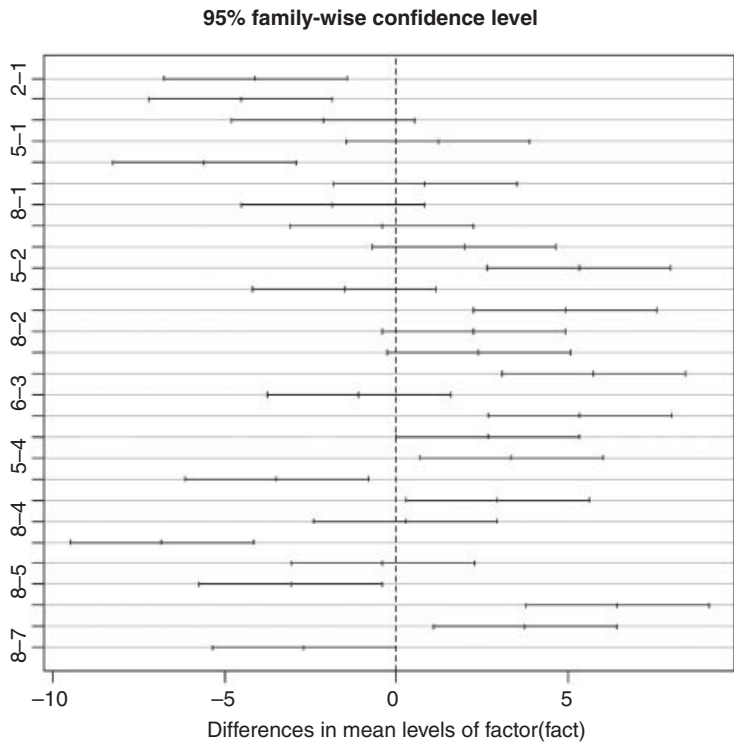
```
model <- aov(response~factor(fact))
summary(model)
```

```
             Df Sum Sq Mean Sq F value Pr(>F)
factor(fact)   7  925.7  132.24   17.48 <2e-16 ***
Residuals    152 1150.1    7.57
```

Indeed, there is compelling evidence ($p<0.0001$) for accepting that there are significant differences between the mean yields of the eight different crop cultivars.

Alternatively, if you want to do multiple comparisons, then because there is no *a priori* way of specifying contrasts between the eight treatments, you might use Tukey's honest significant difference (see p. 531):

```
plot(TukeyHSD(model))
```



Comparisons having intervals that do not overlap the vertical dashed line are significantly different. The vertical dashed line indicates no difference between the mean values for the factor-level comparisons indicated on the *y* axis. Thus, we can say that the contrast between cultivars 8 and 7 (8–7) falls just short of significance (despite the fact that their notches do not overlap; see above), but the comparisons 7–6 and 8–6 are both significant (their *boxes* do not overlap, let alone their notches). The missing comparison labels on the *y* axis of the HSD plot have to be inferred from a knowledge of the number of factor levels (8 in this example). So, since 8 vs. 7 is labelled, the next one up must be 8–6 and the one above that is 7–6, then we find 8–5 labelled, so it must be 7–5 above that and 6–5 above that, then 8–4 labelled, and so on.

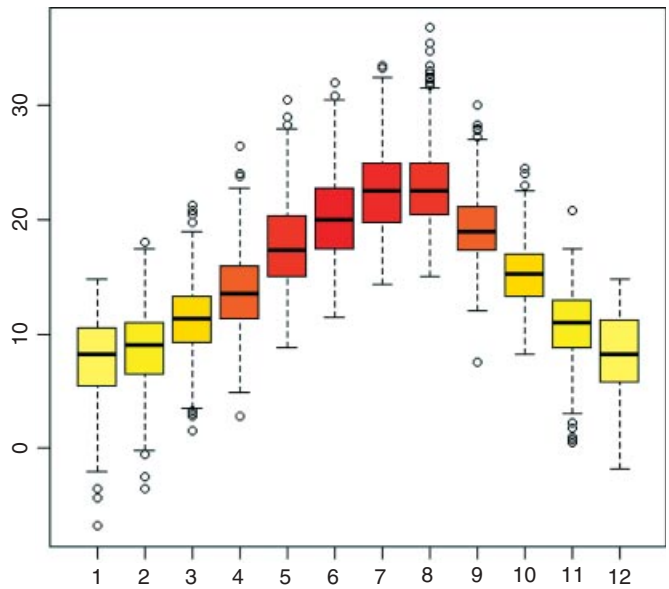
5.6.4 Using colour palettes with categorical explanatory variables

You can create a vector of colours from a palette, then refer to the colours by their subscripts within the palette. The key is to create the right number of colours for your needs. Here, we use the built-in heat.colors to shade the temperature bars in Silwood Weather. We want the colours to grade from cold to hot then back

to cold again from January to December:

```
data <- read.table("c:\\temp\\silwoodweather.txt",header=T)
attach(data)
month <- factor(month)

season <- heat.colors(12)
temp <- c(11,10,8,5,3,1,2,3,5,8,10,11)
plot(month,upper,col=season[temp])
```



Colouring the other parts of the box-and-whisker plot is explained on p. 918.

5.7 Plots for single samples

When we have a just one variable, the choice of plots is more restricted:

- `hist(y)` histograms to show a frequency distribution
- `plot(y)` index plots to show the values of *y* in sequence
- `plot.ts(y)` time series plots
- `pie(x)` compositional plots like pie diagrams

5.7.1 Histograms and bar charts

A common mistake among beginners is to confuse histograms and bar charts. Histograms have the response variable on the *x* axis, and the *y* axis shows the frequency (or the probability density) of different values of the response. In contrast, a bar chart has the response variable on the *y* axis and a categorical explanatory variable on the *x* axis.

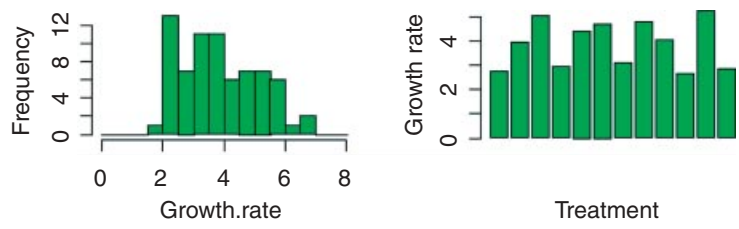
Let us look at an example: the response variable is the growth rate of daphnia in different water qualities; there are four different detergents and three different clones of daphnia.

```
data<-read.table("c:\\temp\\daphnia.txt",header=T)
attach(data)
names(data)
```

```
[1] "Growth.rate" "Water"       "Detergent"   "Daphnia"
```

The histogram shows the frequency with which each growth rate was observed over the experiment as a whole. There are many different bar charts we could draw: here are the mean growth rates cross-classified by clone and detergent:

```
par(mfrow=c(1,2))
hist(Growth.rate,seq(0,8,0.5),col="green",main="")
y <- as.vector(tapply(Growth.rate,list(Daphnia,Detergent),mean))
barplot(y,col="green",ylab="Growth rate",xlab="Treatment")
```

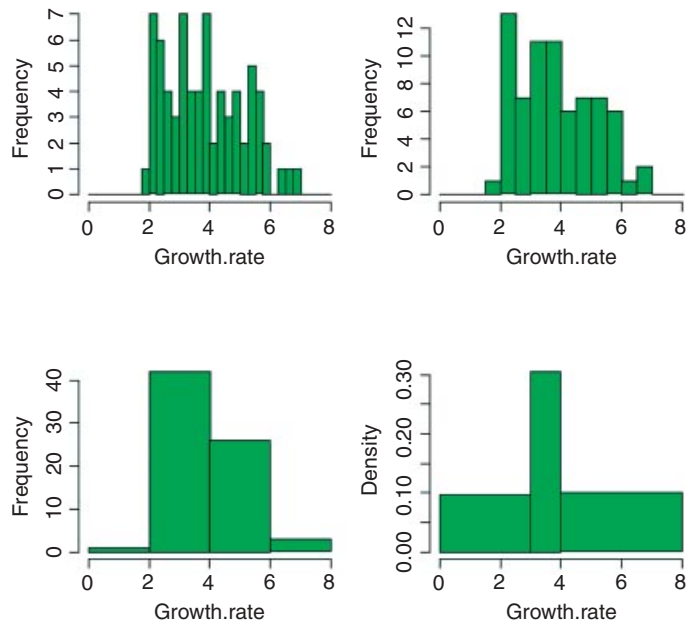


There is a superficial similarity between the two plots in that both have numerous green vertical bars. But there the similarity ends. The histogram on the left has `Growth.rate` on the *x* axis, but the bar plot on the right has `Growth.rate` on the *y* axis. The *y* axis on the histogram shows the count (frequency) of the number of times that values from a given interval of growth rates were observed in the whole experiment. The *y* axis on the bar plot shows the arithmetic mean growth rate for that particular experimental treatment. There is no need to labour the point, but you must be absolutely sure that you understand the difference between a histogram and a bar plot, and try not to refer to a bar chart as a histogram or vice versa.

5.7.2 Histograms

The divisions of the *x* axis into which the values of the response variable are distributed and then counted are called **bins**. Histograms are profoundly tricky, because what you see depends on the subjective judgements of where exactly to put the bin margins. Wide bins produce one picture, narrow bins produce a different picture, unequal bins produce confusion.

```
par(mfrow=c(2,2))
hist(Growth.rate,seq(0,8,0.25),col="green",main="")
hist(Growth.rate,seq(0,8,0.5),col="green",main="")
hist(Growth.rate,seq(0,8,2),col="green",main="")
hist(Growth.rate,c(0,3,4,8),col="green",main="")
```



The bins are 0.25 units wide in the lop left-hand histrogram, 0.5 wide in the top right, 2.0 wide in the bottom left, and there are three different widths (3, 1, then 4) in the bottom right. The narrower the bins, the lower the peak frequencies (note that the *y* scale changes: 7, 12, 40). Small bins produce multimodality (top left), broad bins unimodality (bottom right). When there are different bin widths (bottom right), the default in R is for `hist` to convert the counts (frequencies) into densities (so that the total green area is 1.0).

The convention adopted in R for showing bin boundaries is to employ square and round brackets, so that `[a,b)` means 'greater than or equal to *a* but less than *b*' [square then round), and `(a,b]` means 'greater than *a* but less than or equal to *b*' (round then square]. The point is that it must be unequivocal which bin gets a given number when that number falls exactly on a boundary between two bins. You need to take care that the bins can accommodate both your minimum and maximum values.

The function `cut` takes a continuous vector and cuts it up into bins which you can then use for counting. To show how it works, we shall use `cut` with the daphnia data to produce the density distribution shown above in the bottom right. First, we create a vector of bin edges. To do this, we need to know the range of the growth rates:

```
range(Growth.rate)
```

```
[1] 1.761603 6.918344
```

So a lower bound of 0 and an upper bound of 8 will encompass all of the data. We want edges at 3 and 4, so the vector of bin edges is:

```
edges <- c(0,3,4,8)
```

The next bit is what can seem confusing at first. We create a new vector called `bin` which contains the names of the bins (the factor levels) into which each value of growth rate will be placed. Obviously, this new vector

is the same length as `Growth.rate`. It is a factor with as many levels as there are bins (three in this case). The names of the factor levels indicate the bin margins and the edge convention, indicated by round and square brackets (0,3] in this default case:

```
bin <- cut(Growth.rate,edges)
bin
```

```
 [1] (0,3] (0,3] (3,4] (0,3] (3,4] (4,8] (4,8] (3,4] (4,8] (0,3] (3,4]
[12] (0,3] (3,4] (4,8] (4,8] (4,8] (4,8] (4,8] (0,3] (3,4] (3,4] (3,4]
[23] (3,4] (3,4] (3,4] (4,8] (4,8] (0,3] (0,3] (3,4] (3,4] (4,8] (4,8]
[34] (0,3] (3,4] (4,8] (0,3] (0,3] (3,4] (3,4] (3,4] (4,8] (4,8] (4,8]
[45] (4,8] (3,4] (0,3] (3,4] (4,8] (4,8] (4,8] (3,4] (4,8] (4,8] (0,3]
[56] (3,4] (0,3] (4,8] (4,8] (4,8] (0,3] (3,4] (4,8] (0,3] (0,3] (0,3]
[67] (4,8] (4,8] (4,8] (0,3] (0,3] (0,3]
Levels: (0,3] (3,4] (4,8]
```

```
is.factor(bin)
```

```
[1] TRUE
```

As you can see, the default of the `cut` function is to produce bins with the round bracket on the left and the square bracket on the right: (0,3] (3,4] and (4,8]. This is the option `right = TRUE` (the right-hand value will be *included* in the bin (square bracket), and the left-hand value will appear in the next bin to the left, if one exists). If you want to include the left-hand value in the bin and exclude the right-hand value (as you might with a mapping study), then you need to specify the option `right = FALSE` in the `cut` function (see the example on p. 842). Counting the number of cases in each bin could not be simpler:

```
table(bin)
```

```
bin
(0,3] (3,4] (4,8]
   21    22    29
```

To get the heights of the bars for the density plot we need to allow for the areas of the rectangles. First, the total of the counts,

```
sum(table(bin))
```

```
[1] 72
```

and the relative widths of the bins,

```
diff(edges)
```

```
[1] 3 1 4
```

```
(table(bin)/sum(table(bin)))/diff(edges)
```

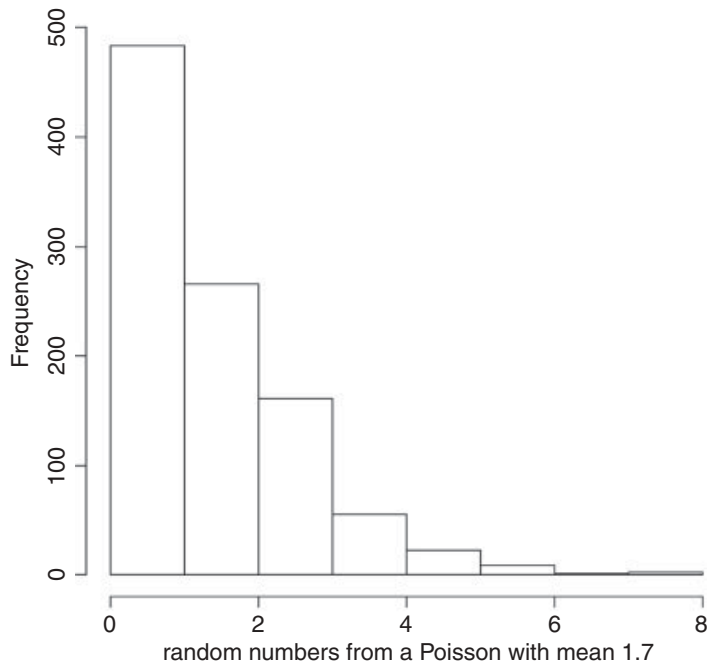
```
bin
     (0,3]      (3,4]      (4,8]
0.09722222 0.30555556 0.10069444
```

These are the heights of the three bars in the density plot (bottom right, above). They do not add to 1 because the bars are of different widths. It is the total area of the three bars that is 1 under this convention.

5.7.3 Histograms of integers

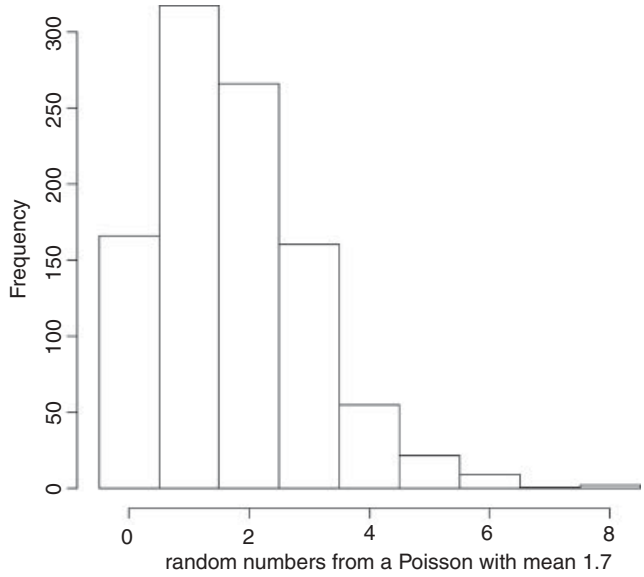
Histograms are excellent for showing the mode, the spread and the symmetry (skew) of a set of data, but the R function `hist` is deceptively simple. Here is a histogram of 1000 random integers drawn from a Poisson distribution with a mean of 1.7. With the default 'pretty' scaling to produce eight bars, the histogram produces a graphic that does not clearly distinguish between the zeros and the ones:

```
values <- rpois(1000,1.70)
hist(values,main="",xlab="random numbers from a Poisson with mean 1.7")
```



With low-value integer data like this, it is much better to specify the bins explicitly, using the `breaks` argument. The most sensible breaks for count data are –0.5 to +0.5 to capture the zeros, 0.5 to 1.5 to capture the 1s, and so on; `breaks=(-0.5:8.5)` generates such a sequence automatically. Now the histogram makes clear that 1s are roughly twice as frequent as zeros:

```
hist(values,breaks=(-0.5:8.5),main="",
            xlab="random numbers from a Poisson with mean 1.7")
```



That's more like it. Now we can see that the mode is 1 (not 0), and that 2s are substantially more frequent than 0s. The distribution is said to be 'skewed to the right' (or 'positively skewed') because the long tail is on the right-hand side of the histogram.

5.7.4 Overlaying histograms with smooth density functions

If it is in any way important, then you should always specify the break points yourself. Unless you do this, the `hist` function may not take your advice about the number of bars or the width of bars. For small-integer data (less than 20, say), the best plan is to have one bin for each value. You create the breaks by starting at –0.5 to accommodate the zeros and going up to $\max(y) + 0.5$ to accommodate the biggest count. Here are 158 random integers from a negative binomial distribution with $\mu = 1.5$ and $k = 1.0$:

```
y <- rnbinom(158,mu=1.5,size=1)
bks <-  -0.5:(max(y)+0.5)
hist(y,bks,main="")
```

To get the best fit of a density function for this histogram we should estimate the parameters of our particular sample of negative binomially distributed counts:

```
mean(y)
```

```
[1]  1.772152
```

```
var(y)
```

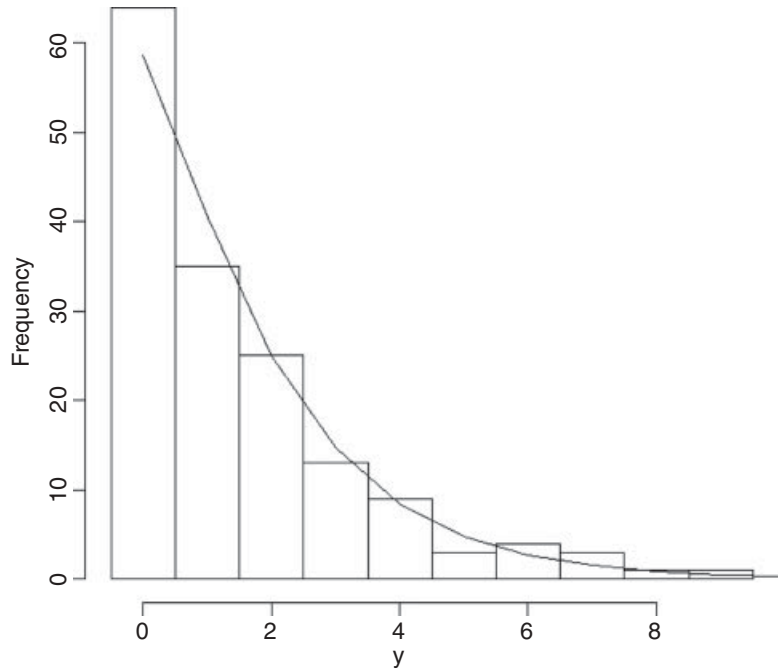
```
[1]  4.228009
```

```
mean(y)^2/(var(y)-mean(y))
```

```
[1]  1.278789
```

In R, the parameter k of the negative binomial distribution is known as `size` and the mean is known as `mu`. We want to generate the probability density for each count between 0 and 11, for which the R function is `dnbinom`:

```
xs <- 0:11
ys <- dnbinom(xs,size=1.2788,mu=1.772)
lines(xs,ys*158)
```



Not surprisingly, since we generated the data, the negative binomial distribution is a very good description of the frequency distribution. The frequency of 1s is a bit low and of 0s is a bit high, but the other frequencies are very well described.

5.7.5 Density estimation for continuous variables

The problems associated with drawing histograms of continuous variables are much more challenging. The subject of density estimation is an important issue for statisticians, and whole books have been written about it (Silverman, 1986; Scott, 1992). You can get a feel for what is involved by browsing the `?density` help window. The algorithm used in `density.default` disperses the mass of the empirical distribution function over a regular grid of at least 512 points, uses the fast Fourier transform to convolve this approximation with a discretized version of the kernel, and then uses linear approximation to evaluate the density at the specified points. The choice of bandwidth is a compromise between smoothing enough to rub out insignificant bumps, and smoothing too much so that real peaks are eliminated. The rule of thumb for bandwidth is

$$b = \frac{\max(x) - \min(x)}{2(1 + \log_2 n)}$$

(where n is the number of data points; for details see Venables and Ripley, 2002). We can compare `hist` with Venables and Ripley's `truehist` for the Old Faithful eruptions data:

```
library(MASS)
attach(faithful)
```

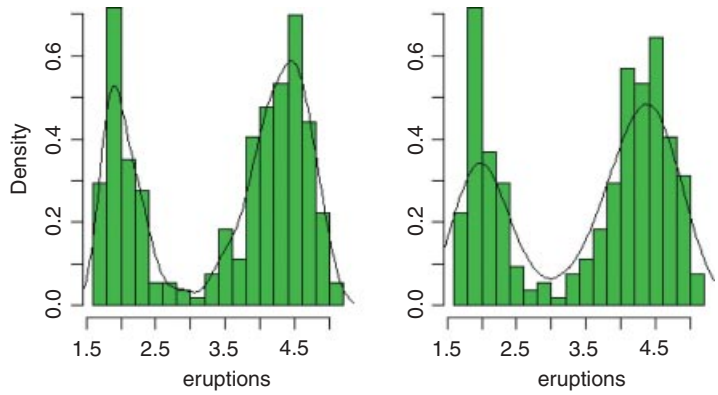
The rule of thumb for bandwidth gives:

```
(max(eruptions)-min(eruptions))/(2*(1+log(length(eruptions),base=2)))
```

```
[1]  0.192573
```

but this produces much too bumpy a fit. A bandwidth of 0.6 looks much better:

```
windows(7,4)
par(mfrow=c(1,2))
hist(eruptions,15,freq=FALSE,main="",col=27)
lines(density(eruptions,width=0.6,n=200))
truehist(eruptions,nbins=15,col=27)
lines(density(eruptions,n=200))
```

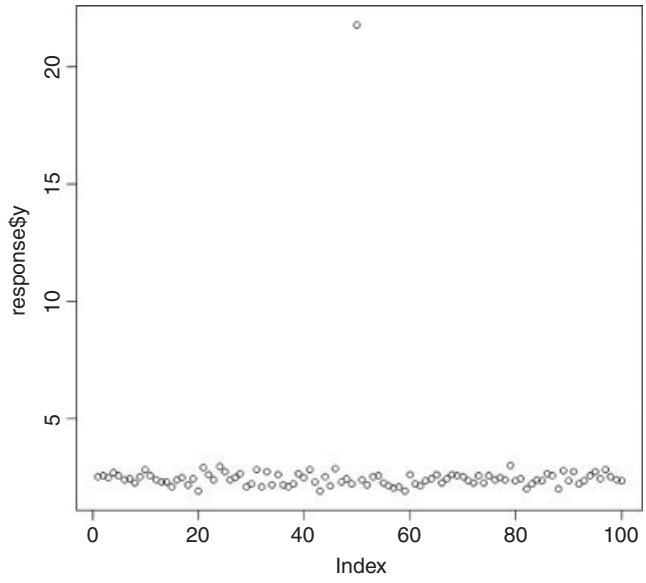


Note that although we asked for 15 bins, we actually got 18. Note also, that although both histograms have 18 bins, they differ substantially in the heights of several of the bars. The left `hist` has two peaks above density = 0.5 while `truehist` on the right has three. There is a sub-peak in the trough of `hist` at about 3.5 but not of `truehist`. And so on. Such are the problems with histograms. Note, also, that the default probability density curve (on the right) picks out the heights of the peaks and troughs much less well than our bandwidth of 0.6 (on the left).

5.7.6 Index plots

The other plot that is useful for single samples is the index plot. Here, `plot` takes a single argument which is a continuous variable and plots the values on the y axis, with the x coordinate determined by the position of the number in the vector (its 'index', which is 1 for the first number, 2 for the second, and so on up to `length(y)` for the last value). This kind of plot is especially useful for error checking. Here is a data set that has not yet been quality checked, with an index plot of `response$y`:

```
response <- read.table("c:\\temp\\das.txt",header=T)
plot(response$y)
```



The error stands out like a sore thumb. We should check whether this might have been a data entry error, such as a decimal point in the wrong place. But which value is it, precisely, that is wrong? What is clear is that it is the only point for which $y > 15$, so we can use the `which` function to find out its index (the subscript within y):

```
which(response$y > 15)
```

```
[1]  50
```

We can then use this value as the subscript to see the precise value of the erroneous y:

```
response$y[50]
```

```
[1]  21.79386
```

Having checked in the lab notebook, it is obvious that this number should be 2.179 rather than 21.79, so we replace the 50th value of y with the correct value:

```
response$y[50] <- 2.179386
```

Now we can repeat the index plot to see if there are any other obvious mistakes

```
plot(response$y)
```

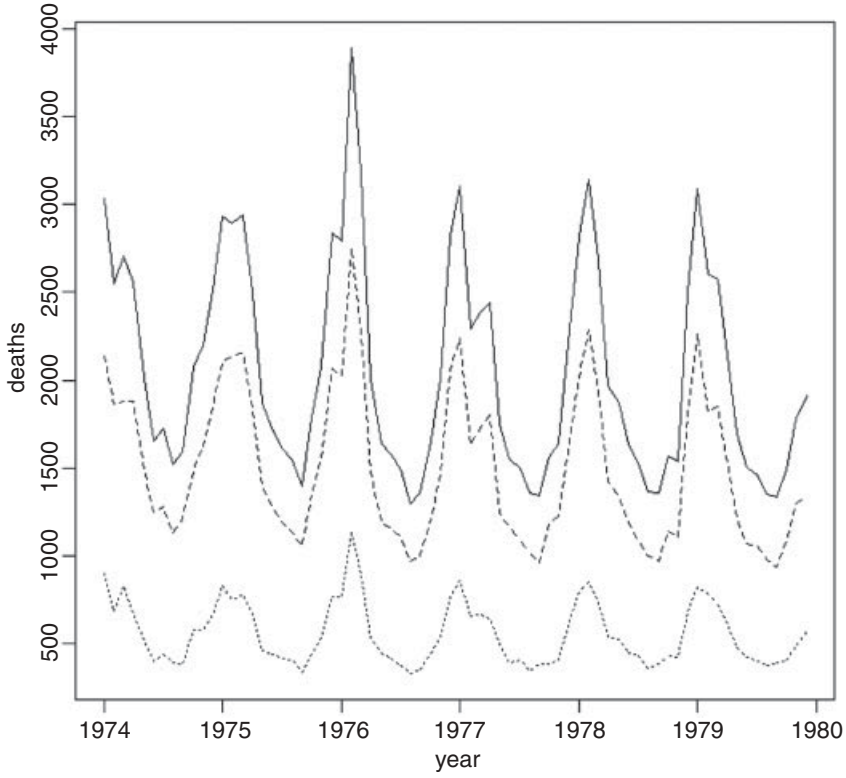
That's more like it.

5.7.7 Time series plots

When a time series is complete, the time series plot is straightforward, because it just amounts to joining the dots in an ordered set of y values. The issues arise when there are missing values in the time series, particularly groups of missing values for which periods we typically know nothing about the behaviour of the time series.

There are two functions in R for plotting time series data: `ts.plot` and `plot.ts`. Here is `ts.plot` in action, producing three time series on the same axes using different line types:

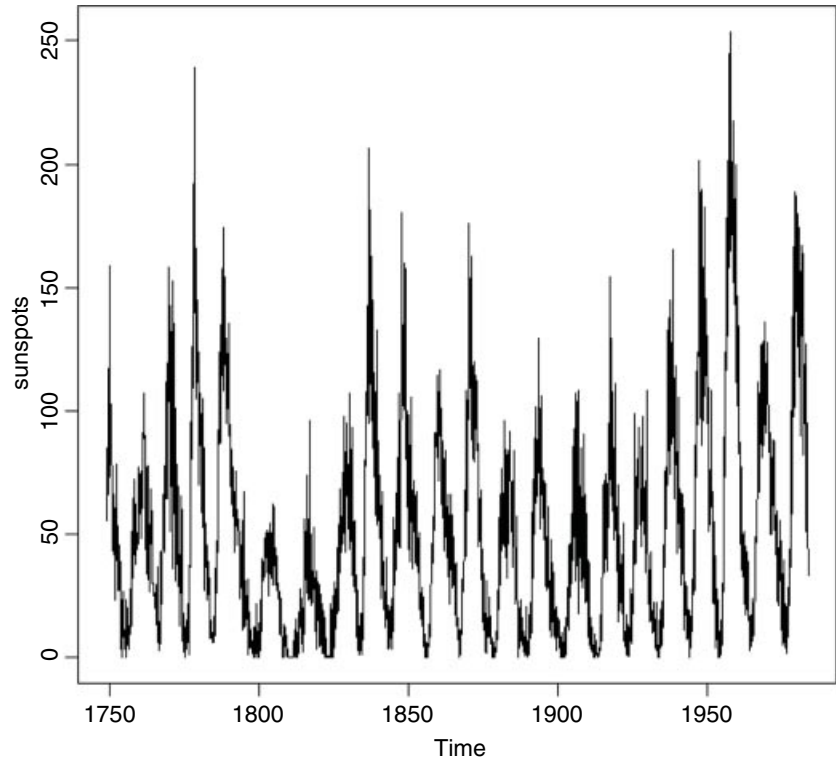
```
data(UKLungDeaths)
ts.plot(ldeaths, mdeaths, fdeaths, xlab="year", ylab="deaths", lty=c(1:3))
```



The upper, solid line shows total deaths, the heavier dashed line shows male deaths and the faint dotted line shows female deaths. The difference between the sexes is clear, as is the pronounced seasonality, with deaths peaking in midwinter.

The alternative function `plot.ts` works for plotting objects inheriting from `class=ts` (rather than simple vectors of numbers in the case of `ts.plot`).

```
data(sunspots)
plot(sunspots)
```



The simple statement `plot(sunspots)` works because sunspots inherits from the time series class, and has the dates for plotting on the *x* axis built into the object:

```
class(sunspots)
```

```
[1]  "ts"
```

```
is.ts(sunspots)
```

```
 [1]  TRUE
```

```
str(sunspots)
```

```
 Time-Series [1:2820] from 1749 to 1984: 58 62.6 70 55.7 85 83.5 94.8  ...
```

5.7.8 Pie charts

Statisticians do not like pie charts because they think that people should know what 50% looks like. Pie charts, however, can sometimes be useful to illustrate the proportional make-up of a sample in presentations. The function `pie` takes a vector of numbers, turns them into proportions, and divides up the circle on the basis of those proportions. It is essential to use a label to indicate which pie segment is which. The label is provided as a vector of character strings, here called `data$names`. Because there are blank spaces in some of the names ('oil shales' and 'methyl clathrates') we cannot use `read.table` with a tab-delimited text file to enter the data. Instead, we save the file called `piedata` as a comma-delimited file, with a '.csv' extention,

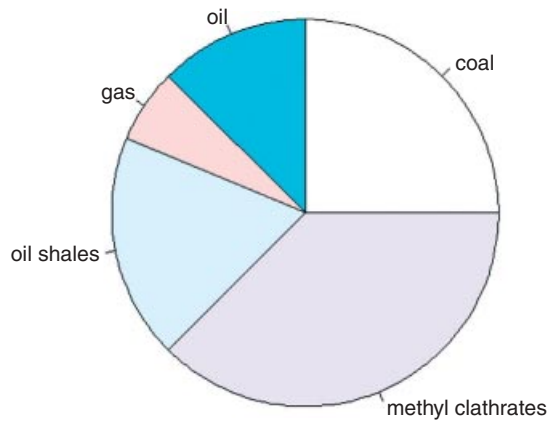
and input the data to R using `read.csv` in place of `read.table`, like this:

```
data <- read.csv("c:\\temp\\piedata.csv")
data
```

```
               names amounts
1               coal       4
2                oil       2
3                gas       1
4        oil shales       3
5 methyl clathrates       6
```

The pie chart is created like this:

```
pie(data$amounts,labels=as.character(data$names))
```

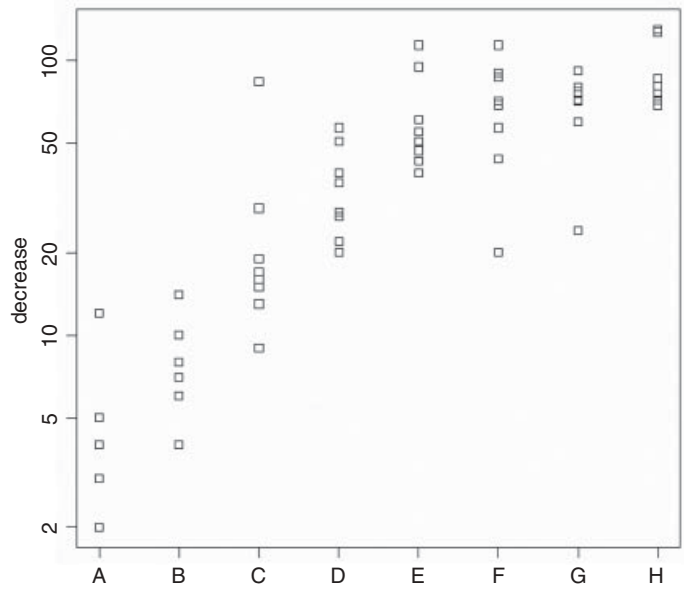


You can change the colours of the segments if you want to (p. 910).

5.7.9 The `stripchart` function

For sample sizes that are too small to use box-and-whisker plots, an alternative plotting method is to use the `stripchart` function. The point of using `stripchart` is to look carefully at the location of individual values within the small sample, and to compare values across cases. The `stripchart` plot can be specified by a model formula `y~factor` and the strips can be specified to run vertically rather than horizontally. Here is an example from the built-in `OrchardSprays` data set where the response variable is called `decrease` and there is a single categorical variable called `treatment` (with eight levels A–H). Note the use of `with` instead of `attach`:

```
data(OrchardSprays)
with(OrchardSprays,
      stripchart(decrease ~ treatment,
      ylab = "decrease", vertical = TRUE, log = "y"))
```



This has the general layout of the box-and-whisker plot, but shows all the raw data values. Note the logarithmic *y* axis, `log = "y"`, and the vertical alignment of the eight strip charts.

5.7.10 A plot to test for normality

Here is a simple function that plots a data set and compares it to a plot of normally distributed data with the same mean and standard deviation:

```
normal.plot <- function(y) {
s <- sd(y)
plot(c(0,3),c(min(0,mean(y)-s * 4*
     qnorm(0.75)),max(y)),xaxt="n",xlab="",type="n",ylab="")
#   for your data's boxes and whiskers, centred at x = 1

top <- quantile(y,0.75)
bottom <- quantile(y,0.25)
w1u <- quantile(y,0.91)
w2u <- quantile(y,0.98)
w1d <- quantile(y,0.09)
w2d <- quantile(y,0.02)
rect(0.8,bottom,1.2,top)
lines(c(0.8,1.2),c(mean(y),mean(y)),lty=3)
lines(c(0.8,1.2),c(median(y),median(y)))
lines(c(1,1),c(top,w1u))
lines(c(0.9,1.1),c(w1u,w1u))
lines(c(1,1),c(w2u,w1u),lty=3)
lines(c(0.9,1.1),c(w2u,w2u),lty=3)
```

```
nou <- length(y[y>w2u])
points(rep(1,nou),jitter(y[y>w2u]))
lines(c(1,1),c(bottom,w1d))
lines(c(0.9,1.1),c(w1d,w1d))
lines(c(1,1),c(w2d,w1d),lty=3)
lines(c(0.9,1.1),c(w2d,w2d),lty=3)
nod <- length(y[y<w2d])
points(rep(1,nod),jitter(y[y<w2d]))

#for the normal box and whiskers, centred at x = 2

n75 <- mean(y)+ s * qnorm(0.75)
n25 <- mean(y)- s * qnorm(0.75)
n91 <- mean(y)+ s * 2* qnorm(0.75)
n98 <- mean(y)+ s * 3* qnorm(0.75)
n9 <- mean(y)- s * 2* qnorm(0.75)
n2 <- mean(y)- s * 3* qnorm(0.75)

rect(1.8,n25,2.2,n75)
lines(c(1.8,2.2),c(mean(y),mean(y)),lty=3)
lines(c(2,2),c(n75,n91))
lines(c(1.9,2.1),c(n91,n91))
lines(c(2,2),c(n98,n91),lty=3)
lines(c(1.9,2.1),c(n98,n98),lty=3)
lines(c(2,2),c(n25,n9))
lines(c(1.9,2.1),c(n9,n9))
lines(c(2,2),c(n9,n2),lty=3)
lines(c(1.9,2.1),c(n2,n2),lty=3)
lines(c(1.2,1.8),c(top,n75),lty=3,col="gray")
lines(c(1.1,1.9),c(w1u,n91),lty=3,col="gray")
lines(c(1.1,1.9),c(w2u,n98),lty=3,col="gray")
lines(c(1.2,1.8),c(bottom,n25),lty=3,col="gray")
lines(c(1.1,1.9),c(w1d,n9),lty=3,col="gray")
lines(c(1.1,1.9),c(w2d,n2),lty=3,col="gray")

# label the two boxes

axis(1,c(1,2),c("data","normal"))   }
```

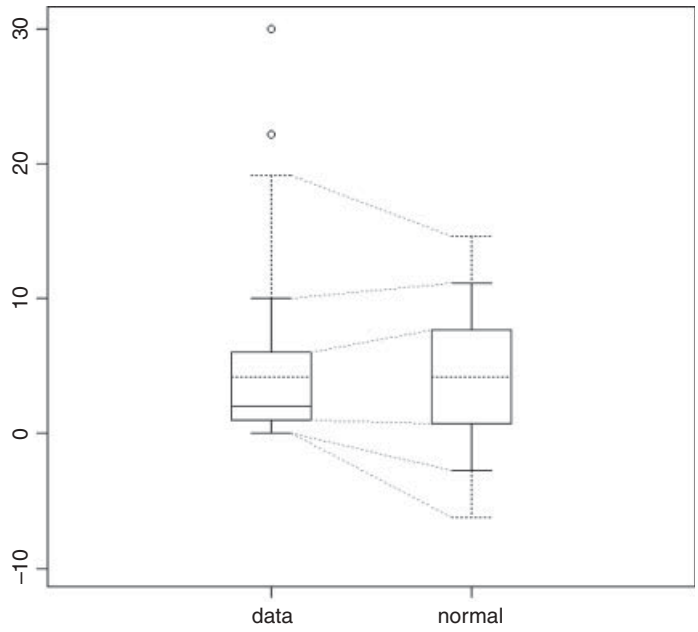
The plots are like extended box-and-whisker plots, in that they show the median inside a box defined by the 25th and 75th percentiles, with solid whiskers running from the 9th percentile to the 91st percentile, and dotted whiskers running to the 2nd and 98th percentiles. Outliers are defined as values outside the 2nd to the 98th percentiles and are plotted as open circles.

Here are our strongly non-normal test data:

```
y <- rnbinom(100,1,0.2)
```

Here is the test:

```
normal.plot(y)
```



Our data (on the left) are non-normal in several obvious ways: the median is lower than the mean (the solid line is below the horizontal dotted line inside the box), the 75th percentile is rather low (the top of the normal box on the right is higher), and our data have two serious outliers (the open circles). Most obviously, however, our data have no negative values, which normally distributed data with a mean and standard deviation as specified would certainly be expected to have (the 9th and 2nd percentiles on the right-hand box are both well below zero, but our minimum value was 0).

5.8 Plots with multiple variables

Initial data inspection using plots is even more important when there are many variables, any one of which might contain mistakes or omissions. The principal plot functions when there are multiple variables are:

- `pairs` for a matrix of scatterplots of every variable against every other;
- `coplot` for conditioning plots where y is plotted against x for different values of z;
- `xyplot` where a set of panel plots is produced.

We illustrate these functions with the ozone data.

5.8.1 The `pairs` function

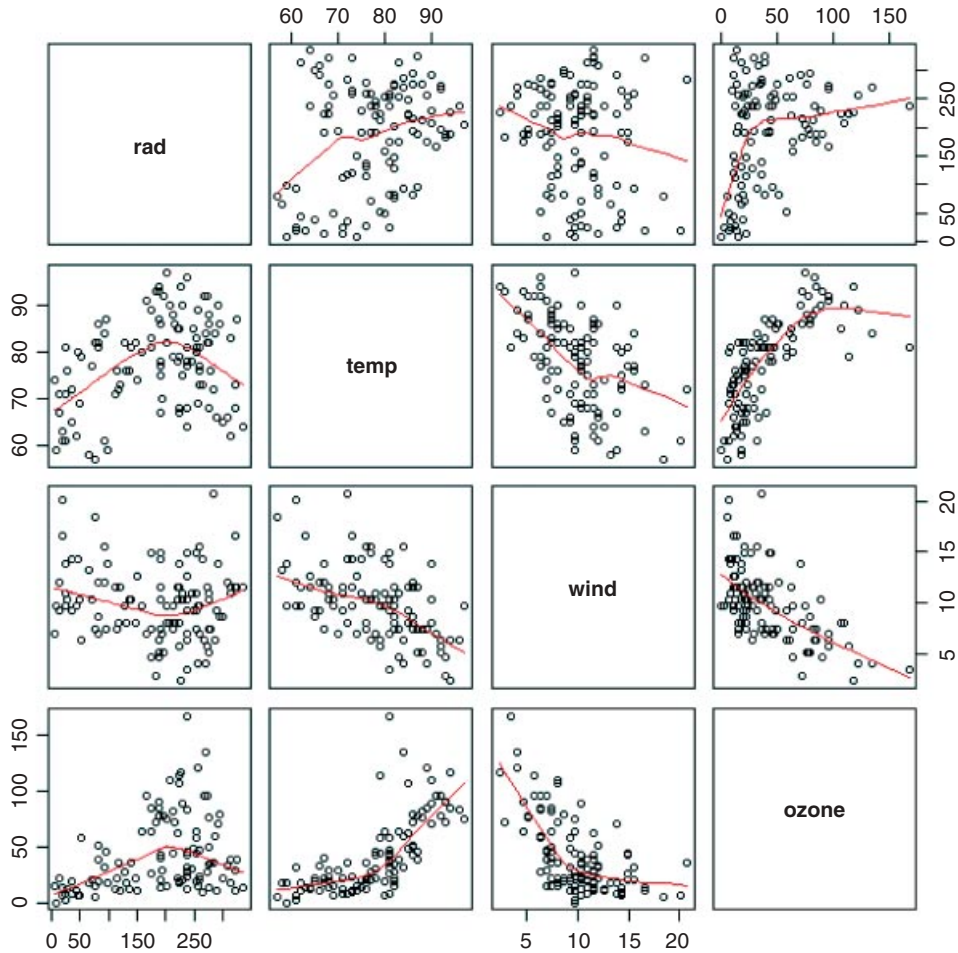
With two or more continuous explanatory variables (i.e. in a multiple regression; see p. 395) it is valuable to be able to check for subtle dependencies between the explanatory variables. The `pairs` function plots every variable in the dataframe on the y axis against every other variable on the x axis: you will see at once what this means from the following example:

```
ozonedata <- read.table("c:\\temp\\ozone.data.txt",header=T)
attach(ozonedata)
names(ozonedata)
```

```
[1]  "rad"  "temp"  "wind"  "ozone"
```

The `pairs` function needs only the name of the whole dataframe as its first argument. We exercise the option to add a non-parametric smoother to the scatterplots:

```
pairs(ozonedata,panel=panel.smooth)
```

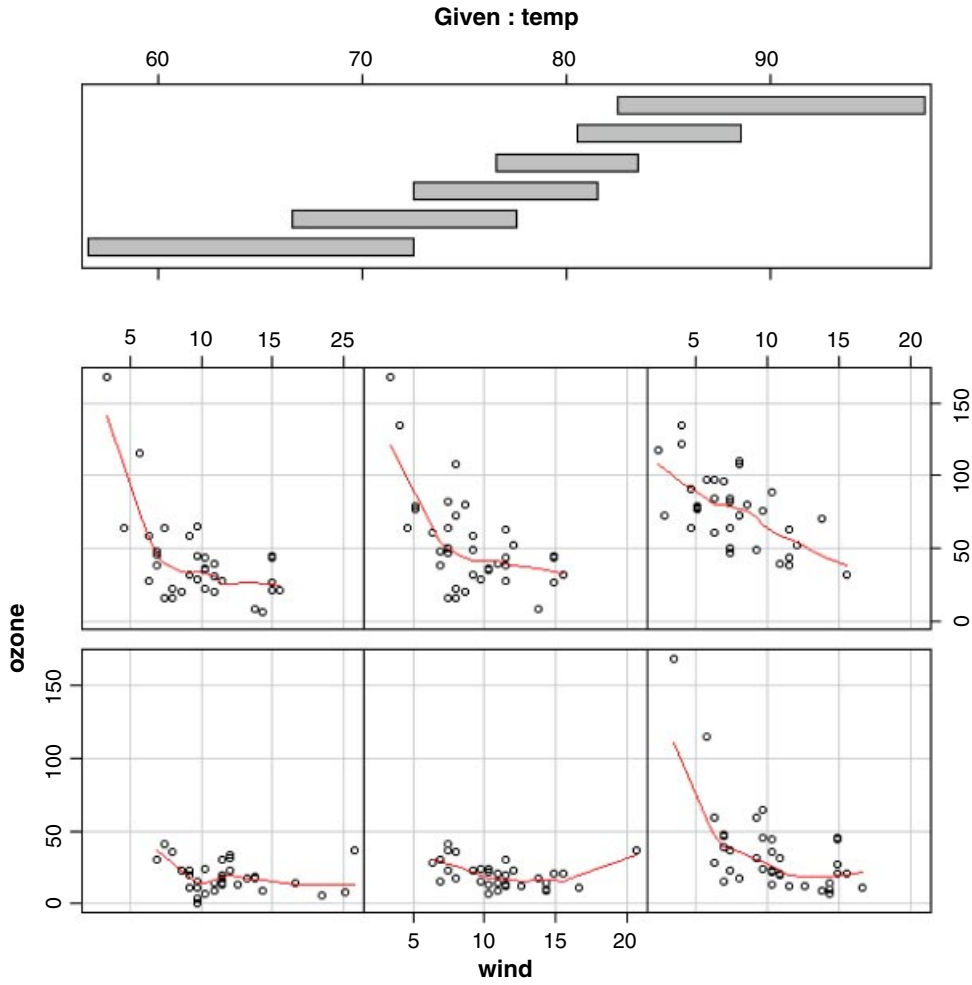


The response variables are named in the rows and the explanatory variables are named in the columns. Thus, in the upper row, labelled `rad`, the response variable (on the *y* axis) is solar radiation. In the bottom row the response variable, `ozone`, is on the *y* axis of all three panels. There appears to be a strong negative non-linear relationship between `ozone` and `wind` speed, a positive non-linear relationship between air temperature and ozone (middle panel in the bottom row) and an indistinct, perhaps humped, relationship between ozone and solar radiation (left-most panel in the bottom row). As to the explanatory variables, there appears to be a negative correlation between wind speed and temperature.

5.8.2 The `coplot` function

A real difficulty with multivariate data is that the relationship between two variables may be obscured by the effects of other processes. When you draw a two-dimensional plot of y against x, then all of the effects of the other explanatory variables are squashed flat onto the plane of the paper. In the simplest case, we have one response variable (ozone) and just two explanatory variables (wind speed and air temperature). The function is written like this:

```
coplot(ozone~wind|temp,panel = panel.smooth)
```



We have the response (`ozone`) on the left of the tilde and the explanatory variable on the x axis (`wind`) on the right, with the conditioning variable after the conditioning operator | (here read as 'given `temp`'). An option employed here is to fit a non-parametric smoother through the scatterplot to emphasize the contrasting trends in each of the panels.

The `coplot` panels are ordered from lower left to upper right, associated with the values of the conditioning variable in the upper panel (`temp`) from left to right. Thus, the lower left-hand plot is for the lowest temperatures (56–72°F) and the upper right plot is for the highest temperatures (82–96°F). This `coplot`

highlights an interesting interaction. At the two lowest levels of the conditioning variable, `temp`, there is little or no relationship between ozone concentration and wind speed, but in the four remaining panels (at higher temperatures) there is a distinct negative relationship between wind speed and ozone. The hard thing to understand about `coplot` involves the 'shingles' that are shown in the upper margin (given `temp` in this case). The overlap between the shingles is intended to show how much overlap there is between one panel and the next in terms of the data points they have in common. In this default configuration, half of the data in a panel is shared with the panel to the left, and half of the data is shared with the panel to the right (`overlap = 0.5`). You can alter the shingle as far as the other extreme, when all the data points in a panel are unique to that panel (there is no overlap between adjacent shingles; `overlap = -0.05`).

5.8.3 Interaction plots

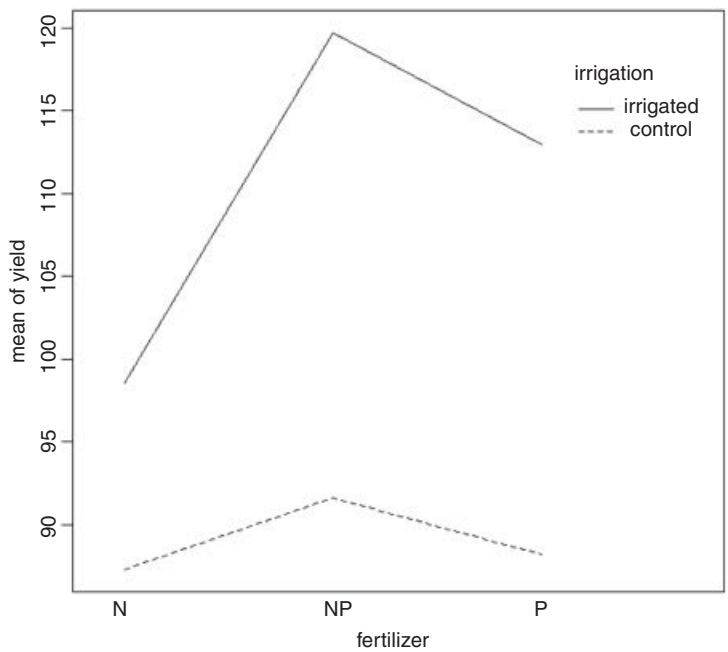
These are useful when the response to one factor depends upon the level of another factor. They are a particularly effective graphical means of interpreting the results of factorial experiments (p. 516). Here is an experiment with grain yields in response to irrigation and fertilizer application:

```
yields <- read.table("c:\\temp\\splityield.txt",header=T)
attach(yields)
names(yields)
```

```
[1]  "yield"  "block"  "irrigation"  "density"  "fertilizer"
```

The interaction plot has a rather curious syntax, because the response variable (`yield`) comes *last* in the list of arguments. The factor listed first forms the *x* axis of the plot (three levels of `fertilizer`), and the factor listed second produces the family of lines (two levels of `irrigation`). The lines join the mean values of the response for each combination of factor levels:

```
interaction.plot(fertilizer,irrigation,yield)
```



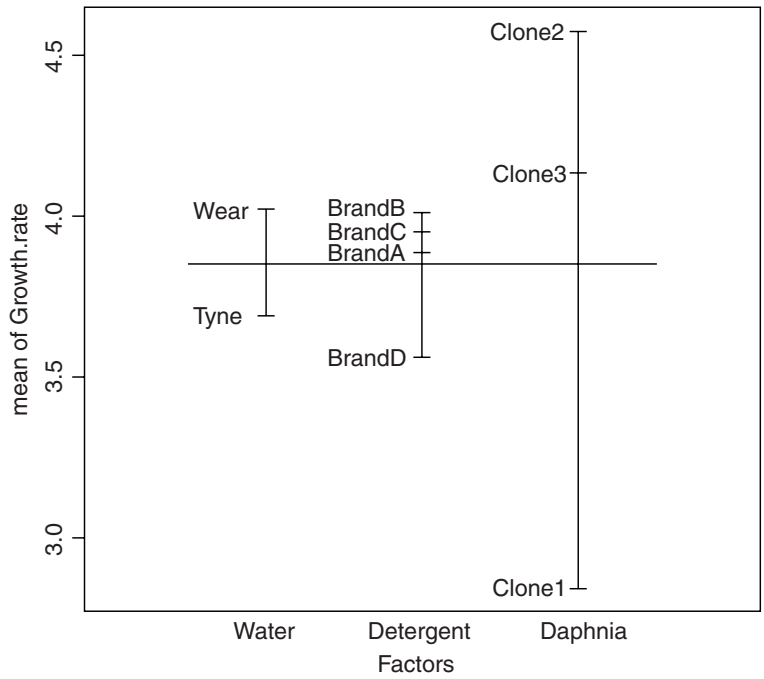
The interaction plot shows that the mean response to fertilizer depends upon the level of irrigation, as evidenced by the fact that the lines are not parallel.

5.9 Special plots

5.9.1 Design plots

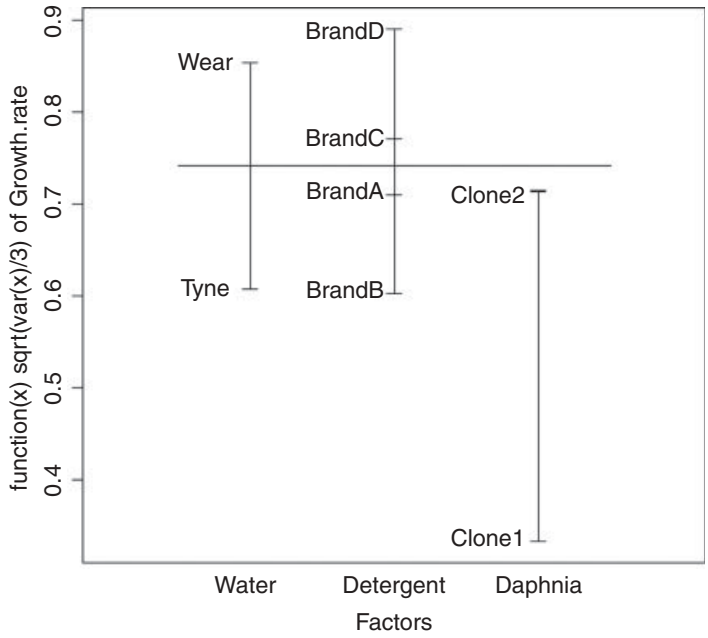
An effective way of visualizing effect sizes in designed experiments is the `plot.design` function which is used just like a model formula:

```
plot.design(Growth.rate~Water*Detergent*Daphnia)
```



This shows the main effects of the three factors, drawing attention to the major differences between the daphnia clones and the small differences between the detergent brands A, B and C. The default (as here) is to plot means, but other functions can be specified such as `median`, `var` or `sd`. Alternatively, you can supply your own anonymous function. Here, for instance, are the standard errors for the different factor levels:

```
plot.design(Growth.rate~Water*Detergent*Daphnia,
                        fun=function(x) sqrt(var(x)/3) )
```



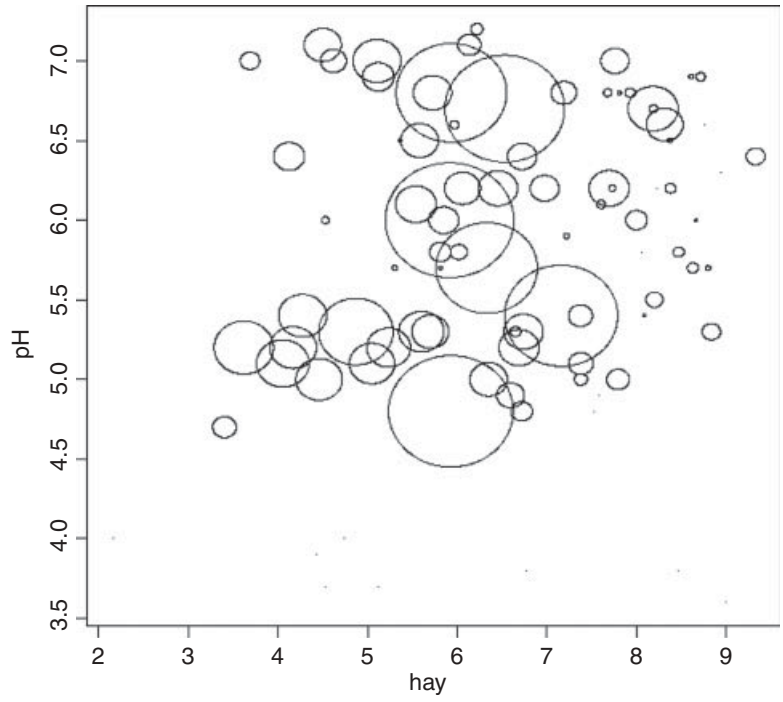
5.9.2 Bubble plots

The bubble plot is useful for illustrating variation in a third variable across different locations in the *x*–*y* plane. Here is a simple function for drawing bubble plots (see also p. 940):

```
bubble.plot <- function(xv,yv,rv,bs=0.1){
      r <- rv/max(rv)
      yscale <- max(yv)-min(yv)
      xscale <- max(xv)-min(xv)
      plot(xv,yv,type="n", xlab=deparse(substitute(xv)),
         ylab=deparse(substitute(yv)))
      for (i in 1:length(xv)) bubble(xv[i],yv[i],r[i],bs,xscale,yscale) }
      bubble <- function (x,y,r,bubble.size,xscale,yscale) {
            theta <- seq(0,2*pi,pi/200)
            yv <- r*sin(theta)*bubble.size*yscale
            xv <- r*cos(theta)* bubble.size*xscale
            lines(x+xv,y+yv) }
```

The example data are on grass yields at different combinations of biomass and soil pH:

```
ddd <- read.table("c:\\temp\\pgr.txt",header=T)
attach(ddd)
names(ddd)
[1]  "FR"  "hay"  "pH"
bubble.plot(hay,pH,FR)
```



In the vicinity of `hay = 6` and `pH = 6` *Festuca rubra* shows one very high value, four intermediate values, two low values and one very low value. Evidently, hay crop and soil pH are not the only factors determining the abundance of *F. rubra* in this experiment.

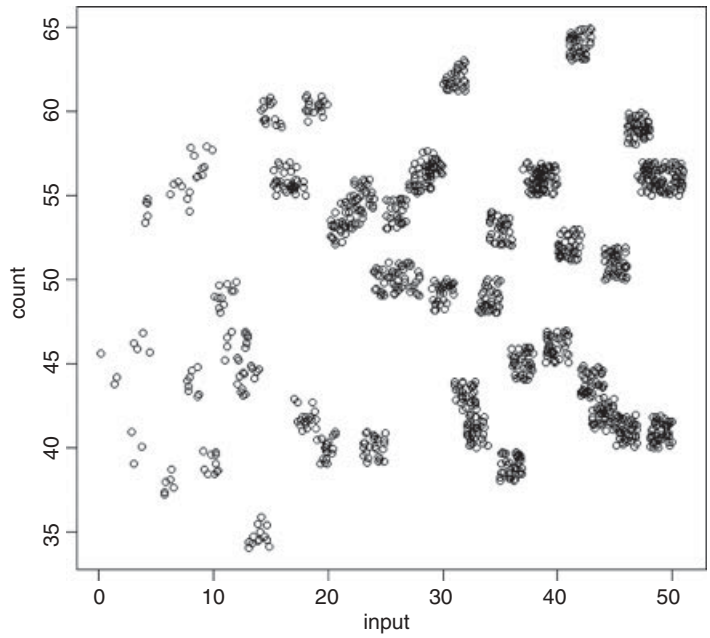
5.9.3 Plots with many identical values

Sometimes, especially with count data, it happens that two or more points fall in exactly the same location in a scatterplot. In such a case, the repeated values of *y* are hidden, one buried beneath the other, and you might want to indicate the number of cases represented at each point on the scatterplot.

```
numbers <- read.table("c:\\temp\\longdata.txt",header=T)
attach(numbers)
names(numbers)
[1]  "xlong"  "ylong"
```

The first option is to 'jitter' the points within the `plot` function. This means to increase or decrease their *x* and/or *y* coordinates by a small random amount until each data point shows separately:

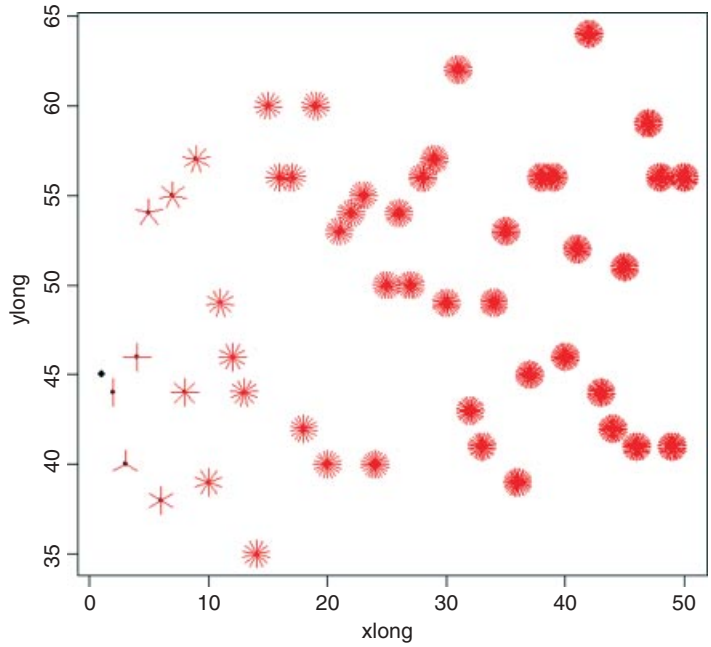
```
plot(jitter(xlong,amount=1),jitter(ylong,amount=1),xlab="input",ylab="count")
```



You need to experiment with the `amount` argument to get the degree of scatter you require (this specifies the limit on the *x* or *y* axis of the amount of jitter on either side of the actual value).

An alternative function is called `sunflowerplot`, so called because it produces one 'petal' of a flower for each value of *y* (if there is more than one) that is located at that particular point. Here it is in action:

```
sunflowerplot(xlong,ylong)
```



As you can see, the replication at each point increases as x increases from 1 on the left to 50 on the right. The petals stop being particularly informative once there are more than about 20 of them (about half way along the x axis). Single values (as on the extreme left) are shown without any petals, while two points in the same place have two petals. As an option, you can specify two vectors containing the unique values of x and y with a third vector containing the frequency of each combination (the number of repeats of each value).

5.10 Saving graphics to file

For publication-quality graphics, you are likely to want to save each of your plots as a PDF or PostScript file. You do this simply by specifying the 'device' before you start plotting, then turning the device off once you have finished. The default device is your computer screen, and you can obtain a rough and ready copy of the graph (press Ctrl + C) which you can then paste into a document outside R (press Ctrl + V).

```
data <- read.table("c:\\temp\\pollute.txt",header=T)
attach(data)
```

You are most likely to want to save to a PDF file. Here is how you do so:

```
pdf("c:\\temp\\pollution.pdf",width=7,height=4)
par(mfrow=c(1,2))
plot(Population,Pollution)
plot(Temp,Pollution)
dev.off()
```

Here is how you save to a PostScript file:

```
postscript("c:\\temp\\pollution.ps",width=7,height=4)
par(mfrow=c(1,2))
plot(Population,Pollution)
plot(Temp,Pollution)
dev.off()
```

There are numerous options for the `pdf` and `postscript` functions, but width and height are the ones you are likely to want to change most often. The sizes are in inches. You can specify any non-default arguments that you want to change (`width, height, onefile, family, title, fonts, paper, encoding, pointsize, bg, fg, pagecentre, useDingbats, colormodel, fillOddEven` and `compress`) using the functions `pdf.options(..., reset = FALSE)` and `ps.options(..., reset = FALSE)` before you invoke either `pdf` or `postscript`. The logical option `reset = TRUE` resets all the options to their default, 'factory-fresh' values. Don't forget to set `dev.off()` once you have finished.

5.11 Summary

It is worth restating the really important things about plotting.

- **Plots**: `plot(x,y)` gives a scatterplot if `x` is continuous, and a box-and-whisker plot if `x` is a factor. Some people prefer the alternative syntax `plot(y~x)` using 'tilde' as in a model formula; one advantage is that this has a `subset` option.

- **Type** of plot: Options include lines `type="l"` or null (axes only) `type="n"`.
- **Lines**: `lines(x,y)` plots a smooth function of `y` against `x` using the `x` and `y` values provided. You might prefer `lines(y~x)`.
- **Line types**: Useful dotted or dashed lines; `lty=2` (an option in `plot` or `lines`).
- **Points**: `points(x,y)` adds another set of data points to a plot. You might prefer `points(y~x)`.
- **Plotting characters** for different data sets: `pch=16` or `pch="*"` (an option in `points` or `plot`).
- **Axes**: setting non-default limits to the *x* or *y* axis scales uses `xlim=c(0,25)` and/or `ylim=c(0,1)` as an option in `plot`.
- **Labels**: use `xlab` and `ylab` to label the *x* and *y* axes.
- Scales: use `ylim` and `xlim` to control the top and bottom values on your axes.