# Chapter 3

# Continuous data

## 3.1 Continuous data

### 3.1.1 Introduction

This chapter deals with simple methods for the analysis of continuous or quantitative data. The distinction is often drawn between *interval-scaled* and *ratio-scaled* data, the latter type involving positive numbers and for which the concept of a ratio is meaningful. Temperatures on the Celsius (centigrade) and Farenheit scales are examples of interval scaled data. For example, we cannot say that a temperature of 40 degrees centigrade is 'twice as hot' (a ratio concept) as one of 20 degrees; using the Farenheit scale converts to 104 and 68 with a ratio 1.53. This differs from 2 and shows that the scales are not ratio-scales. On the other hand it makes sense to say that a pot that is 10 cm tall is twice as tall as one of 5 cm, so height is a ratio-scaled variable. The same is true of many variables used in archaeological data analysis (e.g., length, weight).

Continuous data are usually contrasted with discrete (or counted) data, the subject of Chapter 4. The nature of the data should reflect the method of analysis, including graphical presentation, and this is sometimes neglected. In deciding how to analyze a set of data it should be emphasized that the important issue is whether or not measurements are of a variable that is continuous, *in principle*. For example, (estimated) age at death is a continuous variable but may only be measured to the nearest year. In fact all continuous measurements are inevitably truncated/rounded. The height of a pot may be $10.2683310 \cdots$ cm but, depending on the measuring instrument and the accuracy that is realistic, may be recorded as 10 cm, 10.3 cm or 10.27 cm.

For illustration, unpublished data from Cool (1983) are used. They are the lengths (mm) of 90 copper alloy hairpins from southern Britain, 55 classified as early and 35 as late on archaeological grounds (see Cool, 1990, for a review of the

use of such hairpins). The data are given in Table 3.1.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 54 | 56 | 74 | 84 | 85 | 85 | 87 | 88 | 89 | 90 |
| 90 | 92 | 92 | 92 | 92 | 93 | 93 | 93 | 93 | 93 |
| 94 | 94 | 94 | 95 | 95 | 95 | 96 | 96 | 97 | 97 |
| 97 | 98 | 98 | 100 | 100 | 100 | 100 | 101 | 102 | 103 |
| 104 | 104 | 104 | 104 | 105 | 107 | 108 | 108 | 111 | 115 |
| 115 | 116 | 123 | 128 | 134 | | | | | |
| | | | | | | | | | |
| 51 | 52 | 54 | 56 | 57 | 58 | 60 | 60 | 61 | 62 |
| 62 | 63 | 63 | 63 | 65 | 65 | 66 | 67 | 68 | 68 |
| 70 | 70 | 70 | 70 | 71 | 74 | 75 | 77 | 78 | 78 |
| 80 | 80 | 82 | 82 | 87 | | | | | |

Table 3.1: *Lengths (mm) of Romano-British copper alloy hairpins from southern Britain (Cool,1990). The upper set are early and the lower set late.*

### 3.1.2 Histograms, dotplots and boxplots

Figure 3.1 shows examples of histograms, dotplots and boxplots for the early hairpins data. Much of this will be familiar. An aim here is to demonstrate the ease with which `R` can be used to get publication quality graphs. Code is given either following the figures or in Section 3.2.2. In terms of substantive interpretation the histograms and dotplot are unimodal, with a central concentration of data that tails off, and nothing too distressing in the way of outliers or long tails. The second histogram and dotplot draw attention to two possible outliers to the left that are smaller than the bulk of the data, but not enough to agonize about. We shall loosely refer to such data as (reasonably) *well-behaved*. Data that have a normal distribution are the ideal example of well-behaved data (Section 12.2.1).

A histogram is defined by counts of measurements in cells that have defined lower and upper limits; the default in software packages is to have cells of equal widths (bin-widths), with a default rule applied to determine the number of bins. With equal bin-widths the heights of the bars associated with the bins are proportional to frequencies. A probabilty density scale can be specified if preferred. If a case lies exactly on the boundary between two bins a rule of some kind is applied to determine whether such cases go into the left or right bin. In what follows `RBpins.early` is the name of the data file given to the early hairpins.

The appearance of default histograms can sometimes be improved by increasing the number of bins (i.e. the original is 'oversmoothed'). To specify 20 bins use `hist(RBpins.early, 20)`. This is a guide since aesthetic considerations, namely

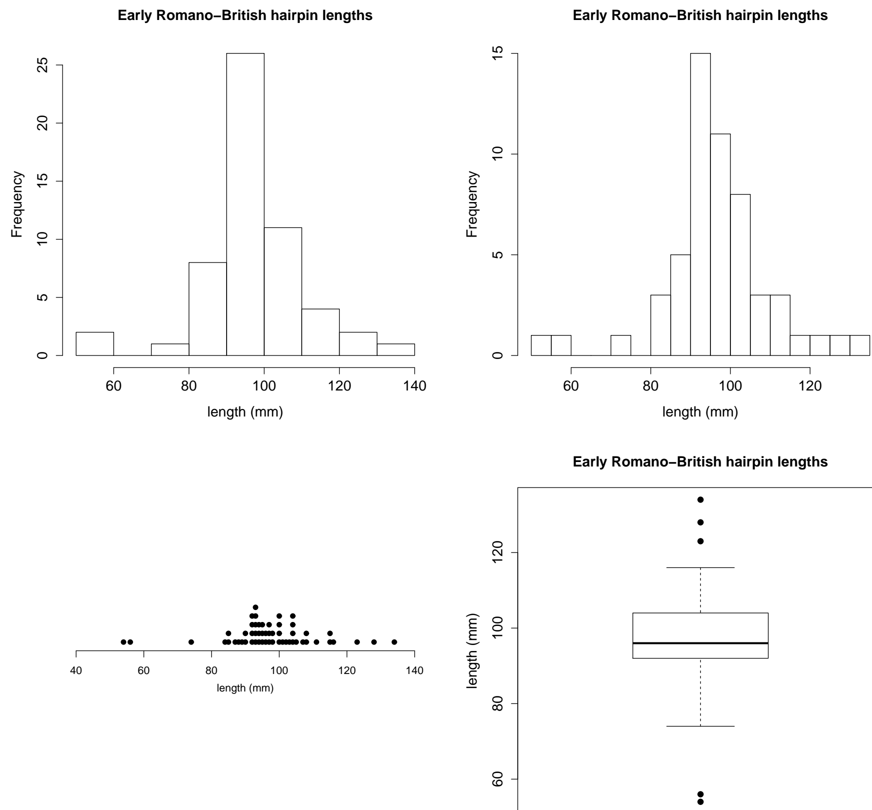


Figure 3.1: *Univariate graphical displays for the lengths of early Romano-British hairpins. The upper left histogram is the* R *default; that to its right specifies 20 bins. A dotplot and boxplot are shown beneath them.*

the desire for 'nice' numbers on the $x$-axis, may dictate a slightly different choice. It is legitimate to play around with the number of bins until the result is judged satisfactory, at which point labeling can be tidied up (Section 2.6.2).

If data are well-behaved it limits complications that may arise in further analysis. In the present instance it legitimizes the use of boxplots (sometimes called box-and-whisker plots) for data display, which are not well-suited to data with more than one mode. The way boxplots are drawn depends on the software used but typically, as here, the box covers the central 50% of the data with its width the interquartile range (IQR), and the line in the box the median. For very well-behaved data whiskers extend to the maximum and minimum of the data. Where, according to a default criterion, which is a bit arbitrary, 'unusual' values (or outliers) are detected the whiskers are broken and the unusual cases highlighted. In

R the default, which can be changed, is to highlight cases more than 1.5 × IQR from the limits of the box.

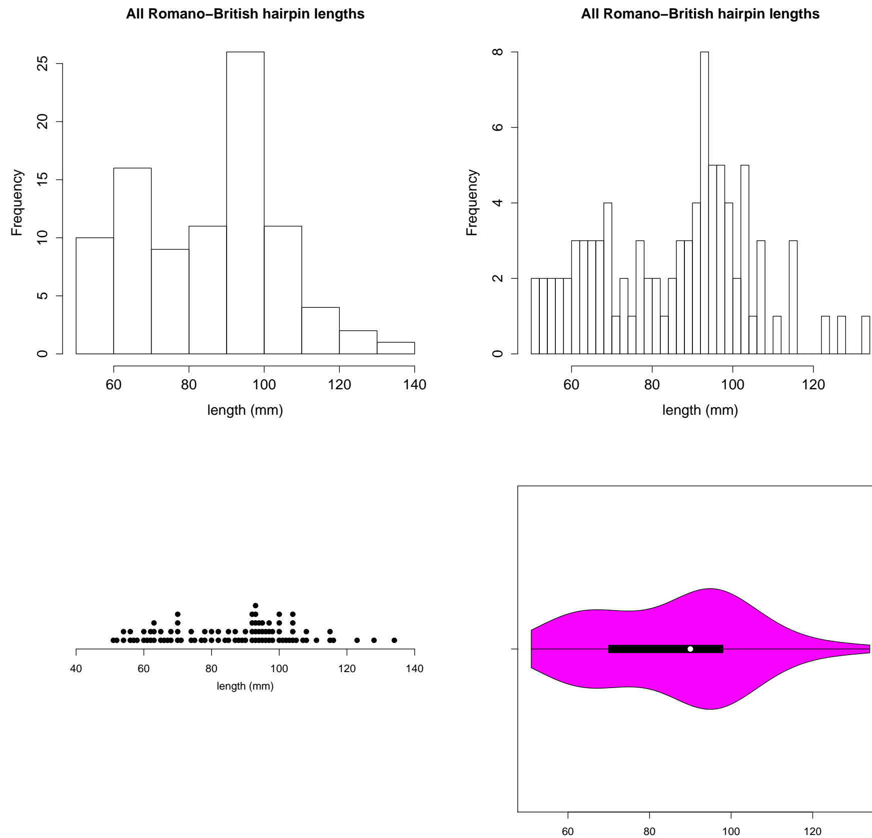


Figure 3.2: *Univariate graphical displays for the lengths of all Romano-British hairpins. See the text for discussion.*

Care needs to be exercised in interpretation, and the present plot exemplifies this. The highlighted cases in the lower part of the boxplot could be interpreted as outliers, whereas those in the upper part are more indicative of a 'slight' tail to the right. In more extreme cases than that illustrated it may be sensible to re-examine the data omitting outliers. If highlighted cases are indicative of a long tail, then the boxplot will not be symmetric. For some methods of statistical analysis long tails are not always welcome, and logarithmic transformation to improve symmetry is common.

Figure 3.2 is mostly as Figure 3.1 but uses all the hairpins, ignoring knowledge of their date. The main difference is the evidence of bimodality in the histograms and dotplot, rendering the boxplot unhelpful. It has been replaced by a violin plot

(Hintze and Nelson, 1998) which, as well as a boxplot, produces an estimate of the density along the boxplot, so that the bimodality can be seen. The package `vioplot` needs to be imported and loaded, with the function of the same name then being used. In the second histogram 30 rather than 20 bins were specified, and it might be thought a little 'bitty'.

### 3.1.3 Kernel density estimates

**Univariate data – one group**

Kernel density estimates (KDEs) would be regarded as mathematically complex by many archaeologists (see Silverman, 1986; Wand and Jones, 1995; Bowman and Azzalini, 1997, to be convinced). Baxter and Beardah (1996) and Baxter *et al.* (1997) provide what counts as early expositions aimed at archaeologists, and Chapter 3 of Baxter (2003) reviews some uses of KDEs in archaeology, to that date. They can now be regarded as a standard method in archaeological data analysis. Conceptually, as illustrated in most of the examples here, KDEs can be used to produce smoothed histograms that overcome many of the problems of the latter illustrated in Whallon (1987). Computationally, `plot(density(RBpins.early))` using the `density` function gets you going for the early hairpins data. We proceed by illustration, starting with Figure 3.3.

The upper-left plot, for the early hairpins data, is the default plot for the `density` function. It is useful to look at because it gives the default bandwidth used. The bandwidth is analogous to the bin-width in a histogram, and controls the appearance of the KDE through the degree of smoothness. Various rules have been put forward for choosing 'good' bandwidths. These are mathematically based and, to my eye, sometimes result in KDEs that are too smooth. As with bin-widths, my preference is to experiment with different bandwidths and select the KDE subjectively. The upper-right plot uses a smaller bandwidth than the default, but it makes little difference to the appearance, with the tails being a bit bumpier. The labeling has also been modified from the default.

The two lower plots are for all the hairpins. That to the left is designed to show the bimodality in the data – remember, at this stage a difference between early and late hairpins is not being assumed. The plot to the right shows a KDE with a larger bandwidth, and therefore smoother.

The next example was provoked by examination of the data for Mg in Table B.1. It illustrates issues of smoothing and data transformation. Examination of the table shows quite clearly that there are regional differences with the values for Region 3 somewhat smaller than for other regions. This is not evident in the default KDE in the upper left of Figure 3.4. To show that Region 3 is separate in a KDE a rather drastic reduction in the bandwidth is needed, as shown in the

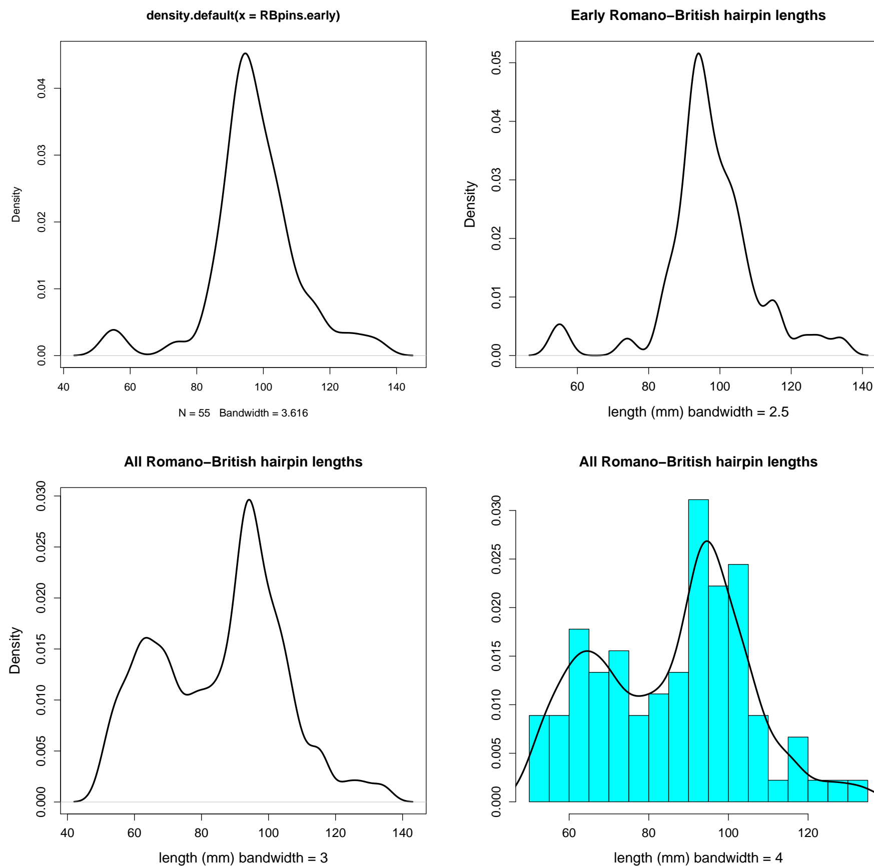


Figure 3.3: *Applications of KDEs to the Romano-British hairpin lengths of Table 3.1, discussed in the text.*

upper-right panel. The bandwidth was chosen to isolate Region 3, and it also does a good job of identifying Region 2 with the highest mode. My immediate reaction on seeing this was that the right-hand side of the plot was undersmoothed; in fact the lower left-hand plot of Figure 2.3 suggests it is reasonable. It is consistent with the fact that there is an extreme case in Region 2 and a small but separate group of five cases.

A common strategy in the analysis of artifactual chemical compositions, where measurements are strictly positive, is to transform to logarithms (Bieber *et al.*, 1976). If this is done the default KDE immediately suggests trimodality, with the mode for Region 3 to the right. Reducing the bandwidth emphasizes the grouping more, without introducing spurious detail (lower figures). The variation in Region 2, noted in the previous paragraph, is not picked up. In these examples some prior knowledge of the data structure, gained from tabular inspection, is needed

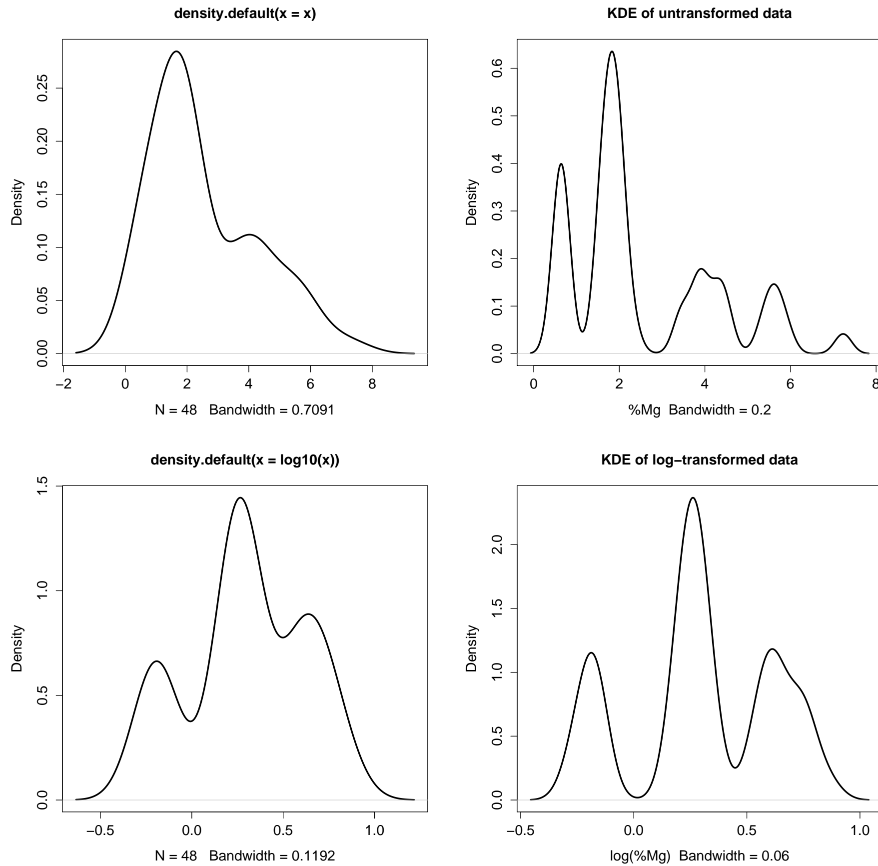


Figure 3.4: *The upper and lower plots contrast KDEs for untransformed and log-transformed data to base 10, for Mg from Table B.1. See the text for a discussion.*

to obtain sensible and informative displays. Almost invariably, it is useful to look at a set of data in more than one way.

## Comparing two groups

The two left-hand panels in Figure 3.5 show histograms for the early and late hairpins. There are arranged vertically and the scales on the $x$-axis and bin-widths have been arranged to be the same. A probability scale is used so the comparison is of shape ignoring sample size. I am not generally a fan of this kind of display, but it works well enough here.

Other examples of this usage are given in Mellars and Wilkinson (1980) to compare the distribution of otolith lengths for samples from late Mesolithic shell-midden sites from Oronsay. Relatively few histograms are used (up to five) and the patterns are sufficiently clear to be informative (though the graphs use a lot of

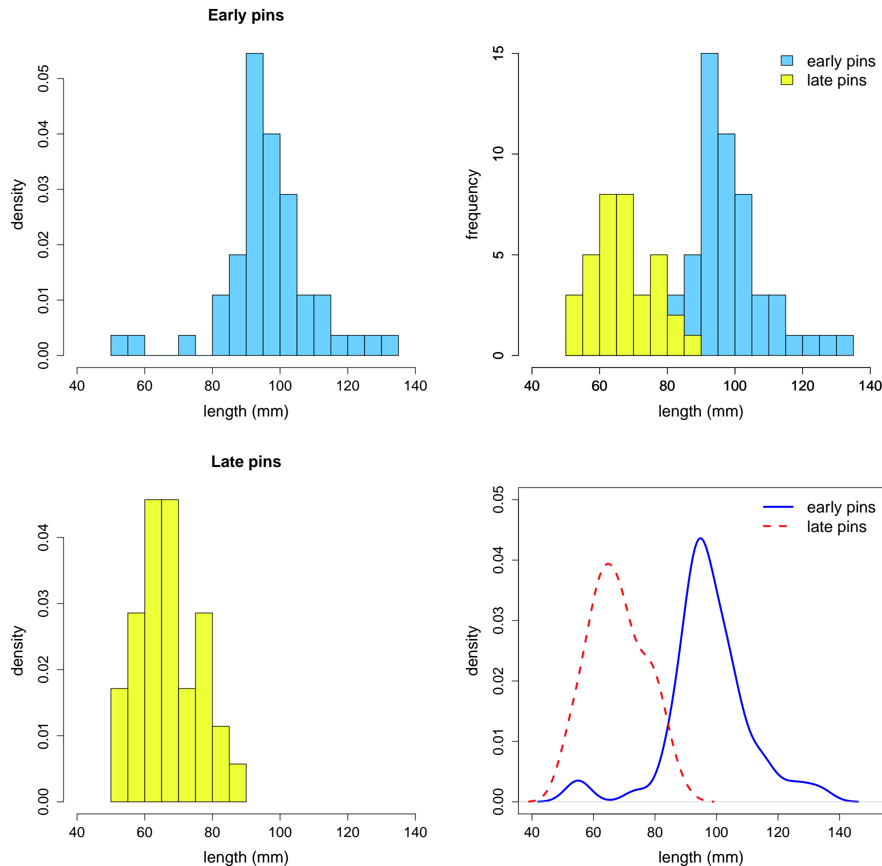


Figure 3.5: *Comparing two groups using histograms and KDEs. For the KDEs the solid and dashed lines are for early and late hairpins respectively. See the text and section notes for a discussion.*

space). I am less convinced by the numerous examples in Albarella *et al.* (2006) where 10/25 pages, most consisting of five histograms, are used to compare mostly pig lower-tooth measurements from a variety of Italian prehistoric contexts. Sample sizes are small for some contexts with, arguably, too many bins used; patterns are not easy to compare; and, with the exception of a clearly bimodal distribution, a table of summary statistics might have served as well or better.

The upper-right plot in Figure 3.5 superimposes the two histograms, and a frequency scale is used which allows the sample size difference to be seen. A problem with such plots is that one histogram will obscure part of the second histogram. In this instance the main message, that the groups have reasonably different locations, is not obscured but, in general, this kind of plot is not really suitable for histograms with much overlap, or for more than two histograms. My

preference in this example would be to superimpose two KDEs, as shown in the lower-right figure. The separation between early and late hairpins is evident, without one KDE obscuring the other. The KDEs are on a density scale so provide no information on sample size differences.

Histograms and KDEs are not the only way of comparing groups. Figure 3.6 shows some possibilities using stripcharts, violin plots and boxplots. The stripchart for the %Fe data from Table B.1, by region, justifies the use of boxplots as there is no evidence of multimodality (also true of the violin plot). The stripchart suggests there are two outliers in Region 1, also suggested by the relevant boxplot. Boxplots for the Romano-British hairpins data are also shown in the lower-right plot and show that the early hairpins tend to be longer than the late hairpins.

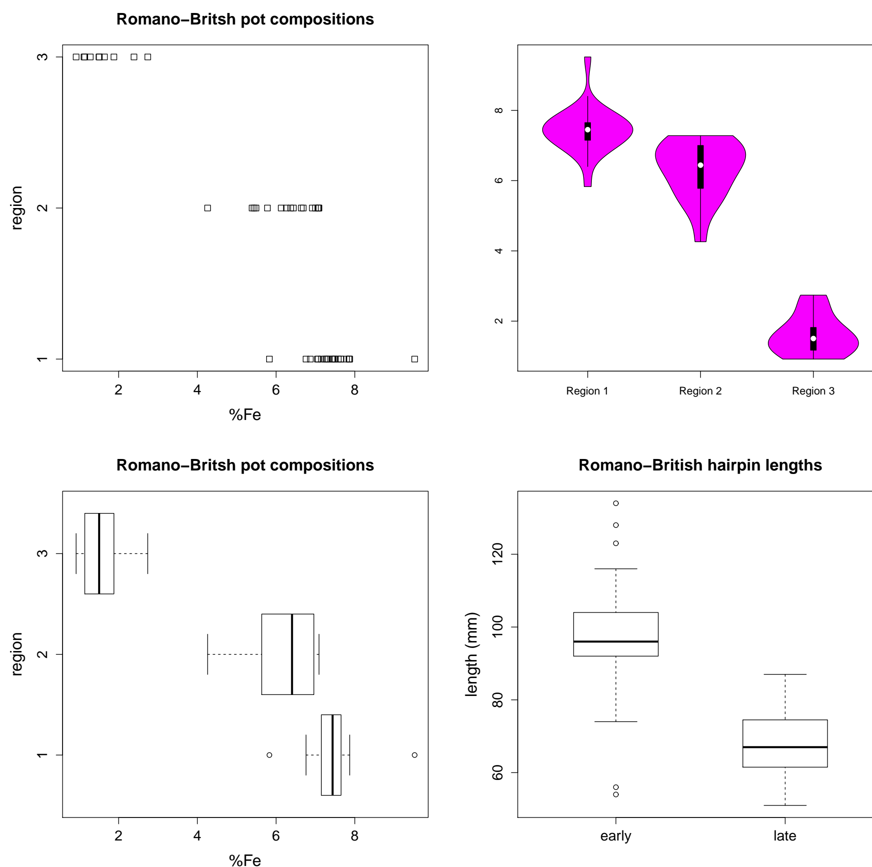


Figure 3.6: *The upper left-hand plot is a 'stripchart' showing the distribution of %Fe by region for the data from Table B.1. To its right, and below, the violin plot and boxplots effect similar comparisons. The final figure is based on the Romano-British hairpins data and shows the distinction between early and late hairpins.*

With software such as R it is sensible to explore different methods of display before selecting one for final publication that tells the story the data deserves. Sometimes the use of more than one display is merited, though publication constraints can militate against this.

## 3.2 R notes

### 3.2.1 Functions

In Section 2.6.3 the possibility of writing a *function* to ease the calculation of the statistics for Table 2.1 was mentioned in passing. User-defined functions are often introduced in the later stages of introductory texts on R, but it is useful to know about them at an early stage. With more than a few lines of code it can be more efficient to write a function to do the job. This can be tested and edited, using the `edit` function, before applying it 'in anger'[1]. One might also wish to reuse a function – for example, by applying the code used to generate the numbers in Table 2.1 for another variable. This will be taken as an initial illustration, then made more general.

The first line in the code that follows names the function `RBpot.statistics1`; `function(x)` defines the function with an argument, `x`, that will be replaced with the data to be used. The function code is enclosed within the braces { and } and consists mainly of code to calculate the statistics previously used. These are collected together in the object `statistics`, where the column bind function, `cbind`, treats each statistic as a $1 \times 1$ table, resulting in a $1 \times 4$ table. Doing it this way retains the statistics' names which makes the output, when printed, easier to read

```
RBpot.statistics1 <- function(x){
Mean <- mean(x)
Median <-median(x)
SD <-  sd(x) # standard deviation
IQR <- IQR(x) # Interquartile Range
statistics <- cbind(Mean, Median, SD, IQR)
list(stats = round(statistics, 2))
}
```

[1]For example, `RBpot.statistics1 <- edit(RBpot.statistics1)` will bring up an edit screen where the function in question can be edited. If mistakes are made then, when you exit from edit mode, you will get a message about this – not always that helpful. Typing the function name will return the original code. To recover the edited version for correction, mistakes and all, use `edit()`.

The `list` function provides the names for and definitions of the objects of interest to be printed – in this case the rounded values of the statistics, named `stats`, are made available (more than one object can be listed). The rounding is achieved by the `round` function which takes the data in the first argument and rounds it to the number of decimal places given in the second argument. The contents of the list can be obtained by typing the function named. Should the output be needed for later use it can be saved using

```
Statistics <- RBpot.statistics1(tubb.data$K)
```

and viewed immediately by typing `Statistics` to get

```
$stats
     Mean  Median  SD   IQR
[1,] 3.18  3.16   0.92  0.96
```

There are several obvious advantages to creating even simple functions like this. One is that other statistics can be added (or subtracted) at will. The argument `x` can be varied to obtain statistics for different subsets of the data or different variables. For example, when invoking the function, replacing `K` with `K[tubb.region == 1]` will produce the statistics for Region 1 only; replacing it with `Fe` will calculate the statistics for that oxide etc.

A natural question to ask is if the regional calculations for all regions can be done with a single function. The following code is one possible way of doing this. The function includes a second argument, `TypeId`, which is where the list containing the group identifiers is entered.

```
RBpot.statistics2 <- function(x, TypeId){
Names <- names(table(TypeId))
Stats <- NULL

for(I in seq(1:length(Names))) {
   Mean <- mean(x[TypeId == Names[I]])
   Median <- median(x[TypeId == Names[I]])
   SD <- sd(x[TypeId == Names[I]]) # standard deviation
   IQR <- IQR(x[TypeId == Names[I]]) # Interquartile Range
  Stats <- rbind(Stats, round(cbind(Mean, Median, SD, IQR), 2))
}
row.names(Stats) <- Names
list(Stats = Stats)
}
```

The `table` function is a convenient way of finding how many categories are represented in the `TypeId` variable and their names are extracted using the `names` function. The extracted variable is here called `Names`, and the subsequent use of the function `length` provides the number of elements in `Names` (three in this instance). Group labels are integers here and will be listed accordingly; with text identifiers the ordering produced by the `table` function corresponds to alphabetical order.

A 'for loop' using the `for` function does the computations. The function `seq`, in conjunction with `length` as used here, generates the number of categories to loop through; each loop creates a table of the statistics with one row using the `cbind` function; and these are bound together using the row bind function, `rbind`, to produce, in this instance, a $3 \times 4$ table of statistics. The binding process needs to start from somewhere and an 'empty' object is created using `Stats <- NULL` that is subsequently filled in during the looping process. Finally, the `row.names` function adds row names to the table so that it is more readable when printed.

The function may be executed and results saved using

```
Statistics <- RBpot.statistics2(tubb.data$K, tubb.region)
```

and printed as in the previous example to obtain

```
$Stats
  Mean Median   SD  IQR
1 3.11   3.13 0.22 0.15
2 4.01   4.28 0.97 0.72
3 2.02   2.03 0.19 0.16
```

As a final example the calculation of statistics for all the variables is illustrated. This is not broken down by region, but this can be achieved, region-by-region, in the call to the function.

```
RBpot.statistics3 <- function(x) {
            Mean <- apply(x, 2, mean)
            Median <- apply(x, 2, median)
            SD <- apply(x, 2, sd)
            IQR <- apply(x, 2, IQR)
            Stats <- rbind(round(cbind(Mean, Median, SD,
                IQR), 2))
        list(Stats = Stats)
    }
```

The `apply` function takes the data matrix `x` as its first argument; the second argument dictates whether calculations are to be based on columns (2 as here)

or rows (1); and the third argument is the function that is to be applied (which can be user-defined). Calling the function as previously illustrated produces the following results.

```
$Stats
    Mean Median   SD  IQR
Al 15.61  16.15 2.70 4.38
Fe  5.83   6.89 2.35 1.93
Mg  2.54   1.93 1.73 2.29
Ca  0.51   0.30 0.45 0.68
Na  0.25   0.21 0.17 0.27
K   3.18   3.16 0.92 0.96
Ti  0.85   0.90 0.21 0.25
Mn  0.08   0.08 0.07 0.05
Ba  0.02   0.02 0.00 0.00
```

If, for example, statistics are needed for Region 1 they can be obtained using

```
RBpot.statistics3(tubb.data[tubb.region == 1,])
```

### 3.2.2 Code used for analyses in the text

*Figures 3.1 and 3.2*

The following function might be used, omitting presentational arguments.

```
somegraphs <- function(x) {
win.graph()
hist(x)
# The default; look at the result and try 20 bins.
win.graph()
hist(x, 20)
library(plotrix) # Needed for the dotplot.
win.graph()
dotplot.mtb(x)
win.graph()
boxplot(x)
}
```

Print the graphs with `somegraphs(RBpins.early)` where `RBpins.early` contains the data on early hairpin lengths. The same graphs can be obtained, without using a function, as

```
hist(RBpins.early)
hist(RBpins.early, 20)
dotplot.mtb(RBpins.early)
boxplot(RBpins.early)
```

but the function can be used for other data, such as `RBpins.late`. In practice presentational arguments are included and it is easier to experiment with these by editing the function rather than re-typing the command every time. This can be done even more easily by expanding the function to add further arguments.

Thus, and for example, to label the axes

```
somegraphs <- function(x, Xlab = " ", Ylab = " ")
```

specifies the arguments;

```
hist(x, xlab = Xlab)
```

is included in the body of the function; and

```
somegraphs(RBpins.early, Xlab = "length (mm)", Ylab = "frequency")
```

labels the $x$- and $y$-axes accordingly.

Several graphs are produced; the `win.graph()` functions ensure that all plots are displayed on the terminal; if omitted only the final graph will be printed. The `hist` function produces the histograms. That to the left is the default histogram. For the second histogram the second argument, 20, specifies the number of bins preferred. To get sensible (i.e. short) labels on the axis `R` may modify this a little.

The `dotplot.mtb` function requires the `plotrix` package to be imported and loaded. It produces a dotplot that imitates what can be obtained in `MINITAB`; it is limited in the control that can be exercised over labeling. The function `stripchart` is the preferred alternative in `R`. The `boxplot` function produces the boxplot. Figure 3.1 has `x = RBpins.early` as its argument; Figure 3.2 replaces `boxplot` with `vioplot` and uses `x = RBpins.all`.

### *Figure 3.3*

```
kde.plots1 <- function(x, y){
library(MASS)     # Needed for ‘truehist’ function.
win.graph(); plot(density(x))
win.graph(); plot(density(x, bw = 2.5))
win.graph(); plot(density(y, bw = 3))
win.graph(), truehist(y, nbins = 20), lines(density(y, bw = 4))
}
kde.plots1(RBpins.early, y = RBpins.all)
```

Presentational aguments are omitted. The semi-colons (;) allow commands to be placed on the same line.

The KDEs are produced using the `density` function . To produce the figure the function arguments, `x` and `y` were `RBpins.early` and `RBpins.all`. The first KDE is the default output for the former and provides a starting point for selecting the bandwidth. This is chosen by a default automatic bandwidth selection procedure; several options are available via the `bw` argument (see `?density` for details). My preference is to choose the bandwidth subjectively after experimentation starting from the default choice and the second plot uses `bw = 2.5` for the early pins data, producing a less smooth estimate.

For all the hairpins `bw = 3` is used in the third plot and `bw = 4` in the final plot, producing greater smoothing. The second of these shows how to overlay the KDE on a histogram using the `lines` function. This requires the `truehist` function from the `MASS` package. The KDE is on a density scale so the histogram must also be on this scale; the `truehist` function provides this by default.

## *Figure 3.4*

The function `log10`, transforms the data to base 10 logarithms; for natural logarithms, $\log_e$, use the `log` function.

```
kde.plots2 <- function(x){
win.graph(); plot(density(x))
win.graph(); plot(density(x, bw = .2))
win.graph(); plot(density(log10(x)))
win.graph(); plot(density(log10(x), bw =.06))
}

kde.plots2(tubb.data$Mg)
```

## *Figure 3.5*

In the following code the `col` argument for the `hist` function shows how to color the bars of the histogram. The `xlim` argument controls the range of the $x$-axis, with `ylim` doing the same for the $y$-axis. There are various reasons one might wish to do this; to restrict the range to 'magnify' parts of a plot, or to expand a plot to accommodate a legend, for example. In the present instance it is used to ensure that histograms have the same range on both the $x$- and $y$-axes. In the plots showing two histograms simultaneously this is essential. Here this is done by superimposing two plots; the first of these produces a histogram for the early hairpins and the histogram for the late hairpins is overlaid on this.

This is done using `par(new = T)` with limits given by `xlim = c(40, 140)` and `ylim = c(0, 15)` to ensure compatibility of the plots. Note that the argument `freq = T` is used in both cases, to provide a histogram on a frequency scale. This is the default and could be omitted; to get a probablity density scale, as in the separate histograms for the two groups, use `freq = F`. This removes the effect of the sample sizes in any comparison of the histograms, which the joint plot retains.

The `par` function can take numerous arguments that allow fine control of the graphics to be exercised. See Section 4.4 of Venables and Ripley (2002) for a discussion of this and some examples; `?par` in `R` lists what arguments are available.

```
KDEHist <- function(x, y) {

win.graph()
hist(RBpins.early, n = 20, col = "skyblue", xlim = c(40, 140), freq = F)

win.graph()
hist(RBpins.early, n = 20, xlim = c(40, 140), ylim = c(0, 15), freq = T)

par(new = T)   # This superimposes the plot on the previous one
hist(RBpins.late, n = 10, xlim = c(40, 140), ylim = c(0, 15), freq = T)

legend("topright", c("early pins", "late pins"), fill =
c("skyblue", "yellow"), bty = "n", cex = 1.5)

win.graph()
hist(RBpins.late, col = "yellow", n = 10, xlim = c(40, 140), freq = F)

win.graph()
plot(density(RBpins.early, bw= 4),xlim = c(40, 150), ylim = c(0,.05))
lines(density(RBpins.late, bw= 4))
}

KDEHist(RBpins.early, RBpins.late)
```

### *Figure 3.6*

Most of the features used below have already been discussed and illustrated in the figures in the text. This is the first use of the `stripchart` function, mentioned earlier as an alternative to the dotplot produced by `dotplot.mtb`. The `vioplot` function has fewer graphical capabilities than the other functions used.

The default in the `boxplot` function is to display the plots in a vertical array; the argument `horizontal = T` produces a horizontal array. The `boxwex` argument

controls the width of the boxes and may be used to produce a more appealing appearance of the plot.

```
graphs <- function() {
library(vioplot)     # Needed for the ‘vioplot’ function

win.graph(); stripchart(tubb.data$Fe ~ tubb.region)

win.graph()
vioplot(tubb.data[1:21,2], tubb.data[22:38,2], tubb.data[39:48,2],
names = c("Region 1", "Region 2", "Region 3"))

win.graph()
boxplot(tubb.data$Fe ~ tubb.region, horizontal = T)

win.graph()
boxplot(RBpins.early, RBpins.late, boxwex = .5)
}

graphs()
```