1

Introduction

The traditional graphics subsystem in R is very flexible when it comes to producing standard statistical graphics. It provides a collection of high-level plotting functions that produce entire coherent displays, several low-level routines to enhance such displays and provide finer control over the various elements that make them up, and a system of parameters that allows global control over defaults and other details. However, this system is not very proficient at combining multiple plots in a page. It is quite straightforward to produce such plots; however, doing so in an effective manner, with properly coordinated scales, aspect ratios, and labels, is a fairly complex task that is difficult even for the experienced R user. Trellis graphics, originally implemented in S, was designed to address this shortcoming. The lattice add-on package provides similar capabilities for R users.

The name "Trellis" comes from the trellislike rectangular array of panels of which such displays often consist. Although Trellis graphics is typically associated with multiple panels, it is also possible to create single-panel Trellis displays, which look very much like traditional high-level R plots. There are subtle differences, however, mostly stemming from an important design goal of Trellis graphics, namely, to make optimum use of the available display area. Even single-panel Trellis displays are usually as good, if not better, than their traditional counterparts in terms of default choices. Overall, Trellis graphics is intended to be a more mature substitute for traditional statistical graphics in R. As such, this book assumes no prior knowledge of traditional R graphics; in fact, too much familiarity with it can be a hindrance, as some basic assumptions that are part and parcel of traditional R graphics may have to be unlearned. However, there are many parallels between the two: both provide high-level functions to produce comprehensive statistical graphs, both provide fine control over annotation and tools to augment displays, and both employ a system of user-modifiable global parameters that control the details of the display. This chapter gives a preview of Trellis graphics using a few examples; details follow in later chapters.

1.1 Multipanel conditioning

For the examples in this chapter, we make use of data on the 1997 A-level chemistry examination in Britain. The data are available in the mlmRev package, and can be loaded into R using the `data()` function.

```
> data(Chem97, package = "mlmRev")
```

A quick summary of the A-level test scores is given by their frequency table[1]

```
> xtabs(~ score, data = Chem97)
score
   0    2    4    6    8   10
3688 3627 4619 5739 6668 6681
```

Along with the test scores of 31,022 students, the dataset records their gender, age, and average GCSE score, which can be viewed as a pre-test achievement score. It additionally provides school and area-level information, which we ignore. In this chapter, we restrict ourselves to visualizations of the distribution of one continuous univariate measure, namely, the average GCSE score. We are interested in understanding the extent to which this measure can be used to predict the A-level chemistry examination score (which is a discrete grade with possible values 0, 2, 4, 6, 8, and 10).

1.1.1 A histogram for every group

One way to learn whether the final A-level score (the variable `score`) depends on the average GCSE score (`gcsescore`) is to ask a slightly different question: is the distribution of `gcsescore` different for different values of `score`? A popular plot used to summarize univariate distributions is the histogram. Using the lattice package, which needs to be attached first using the `library()` function, we can produce a histogram of `gcsescore` for each `score`, placing them all together on a single page, with the call

```
> library("lattice")
> histogram(~ gcsescore | factor(score), data = Chem97)
```

which produces Figure 1.1. There are several important choices made in the resulting display that merit attention. Each histogram occupies a small rectangular area known as a *panel*. The six panels are laid out in an array, whose dimensions are determined automatically. All panels share the same scales, which make the distributions easy to compare. Axes are annotated with tick marks and labels only along the boundaries, saving space between panels. A *strip* at the top of each panel describes which value of `score` that panel represents. These features are available in all Trellis displays, and are collectively

[1] Throughout this book, we make casual use of many R functions, such as `data()` and `xtabs()` here, without going into much detail. We expect readers to make use of R's online help system to learn more about functions that are unfamiliar to them.

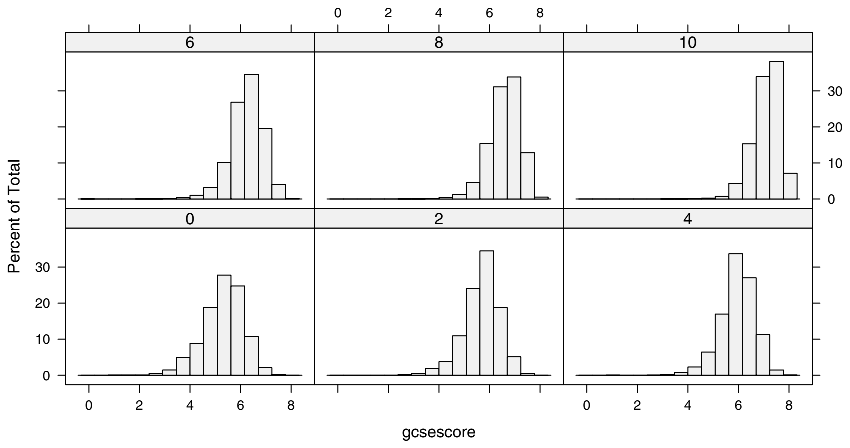


Figure 1.1. A conditional histogram using data on students attempting the A-level chemistry examination in Britain in 1997. The x-axis represents the average GCSE score of the students, and can be viewed as a prior achievement variable. The different panels represent subsets of students according to their grade in the A-level examination, and may be viewed as the response. Strips above each panel indicate the value of the response.

known as *multipanel conditioning*. These choices are intended to make the default display as useful as possible, but can be easily changed. Ultimate control rests in the hands of the user.

1.1.2 The Trellis call

Let us take a closer look at the `histogram()` call. As the name suggests, the `histogram()` function is meant to create histograms. In the call above, it has two arguments. The first (unnamed) argument, `x`, is a *"formula"* object that specifies the variables involved in the plot. The second argument, `data`, is a data frame that contains the variables referenced in the formula `x`.

The interpretation of the formula is discussed in more generality later, but is important enough to warrant some explanation here. In the formula

```
~ gcsescore | factor(score)
```

`factor(score)` (the part after the vertical bar symbol) is the *conditioning variable*, indicating that the resulting plot should contain one panel for each of its unique values (levels). The inline conversion to a factor is related to how the value of the conditioning variable is displayed in the strips; the reader is encouraged to see what happens when it is omitted. There can be more than one conditioning variable, or none at all.

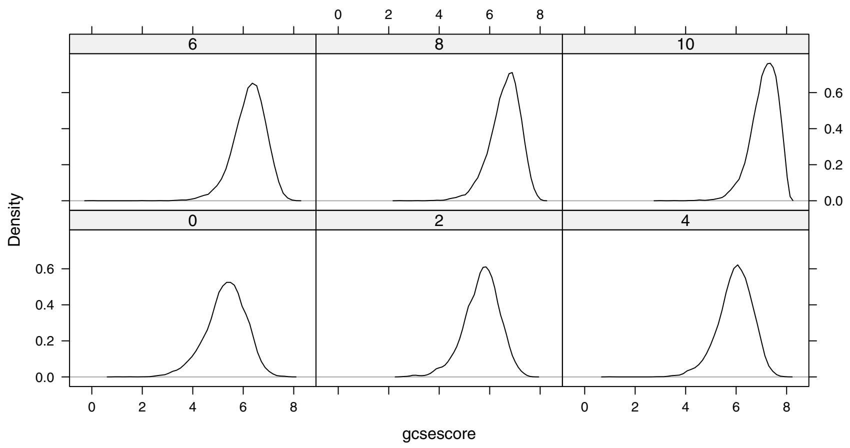


Figure 1.2. Conditional density plots. The data and design are the same as those in Figure 1.1, but histograms are replaced by kernel density estimates.

The part of the formula to the left of the conditioning symbol `|` specifies the *primary variable* that goes inside each panel; `gcsescore` in this case. What this part of the formula looks like depends on the function involved. All the examples we encounter in this chapter have the same form.

1.1.3 Kernel density plots

Histograms are crude examples of a more general class of univariate data summaries, namely, density estimates. `densityplot()`, another high-level function in the lattice package, can be used to graph kernel density estimates. A call that looks very much like the previous `histogram()` call produces Figure 1.2.

```
> densityplot(~ gcsescore | factor(score), data = Chem97,
              plot.points = FALSE, ref = TRUE)
```

There are two more arguments in this call: `ref`, which adds a reference line at 0, and `plot.points`, which controls whether in addition to the density, the original points will be plotted. Displaying the points can be informative for small datasets, but not here, with each panel having more than 3000 points. We show later that `ref` and `plot.points` are not really arguments of `densityplot()`, but rather of the default panel function, responsible for the actual plotting inside each panel.

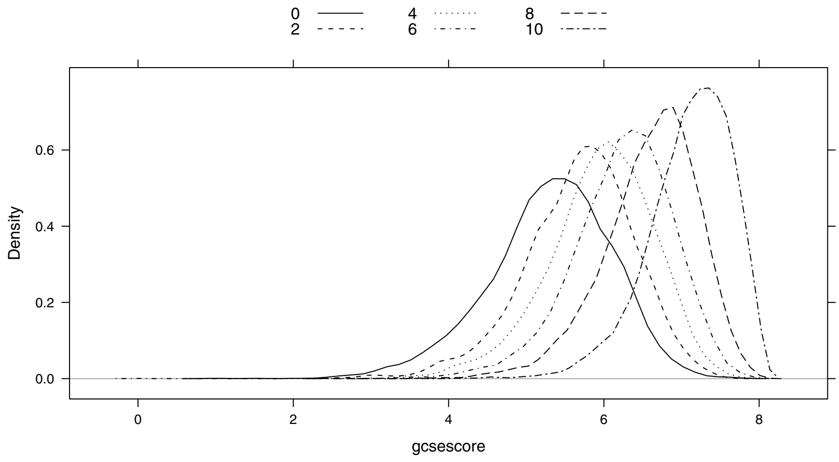


Figure 1.3. Grouped density plots. The density estimates seen in Figure 1.2 are now superposed within a single panel, forcing direct comparison. A legend on the top describes the association between levels of the grouping variable (`score` in this case) and the corresponding line parameters.

1.2 Superposition

Figures 1.1 and 1.2 both show that the distribution of `gcsescore` is generally higher for higher `score`. This pattern would be much easier to judge if the densities were superposed within the same panel. This is achieved by using `score` as a *grouping variable* instead of a conditioning variable[2] in the following call, producing Figure 1.3.

```
> densityplot(~ gcsescore, data = Chem97, groups = score,
              plot.points = FALSE, ref = TRUE,
              auto.key = list(columns = 3))
```

The `auto.key` argument automatically adds a suitable legend to the plot. Notice that it was not necessary to convert `score` into a factor beforehand; this conversion is done automatically. Another important point is that just as with variables in the formula, the expression specified as the `groups` argument was also evaluated in `Chem97` (the `data` argument). This is also true for another special argument, `subset`, which we learn about later.

An important theme in the examples we have seen thus far is the abstraction used in specifying the structure of a plot, which is essentially defined by the *type* of graphic (histogram, density plot) and the *role* of the variables involved (primary display, conditioning, superposition). This abstraction is fundamental in the lattice paradigm. Of course, calls as simple as these will not always suffice in real life, and lattice provides means to systematically

[2] This distinction between grouping and conditioning variables is specific to graphs.

control and customize the various elements that the graphic is comprised of, including axis annotation, labels, and graphical parameters such as color and line type. However, even when one ends up with a seemingly complex call, the basic abstraction will still be present; that final call will be typically arrived at by starting with a simple one and incrementally modifying it one piece at a time.

1.3 The *"trellis"* object

Most regular R functions do not produce any output themselves; instead, they return an object that can be assigned to a variable, used as arguments in other functions, and generally manipulated in various ways. Every such object has a class (sometimes implicit) that potentially determines the behavior of functions that act on them. A particularly important such function is the generic function `print()`, which displays any object in a suitable manner. The special property of `print()` is that it does not always have to be invoked explicitly; the result of an expression evaluated at the top level (i.e., not inside a function or loop), but not assigned to a variable, is printed automatically. Traditional graphics functions, however, are an exception to this paradigm. They typically do not return anything useful; they are invoked for the "side effect" of drawing on a suitable graphics device.

High-level functions in the lattice package differ in this respect from their traditional graphics analogues because they do not draw anything themselves; instead, they return an object, of class *"trellis"*. An actual graphic *is* created when such objects are "printed" by the `print()` method for objects of this class. The difference can be largely ignored, and lattice functions used just as their traditional counterparts (as we have been doing thus far), only because `print()` is usually invoked automatically. To appreciate this fact, consider the following sequence of commands.

```
> tp1 <- histogram(~ gcsescore | factor(score), data = Chem97)
> tp2 <-
      densityplot(~ gcsescore, data = Chem97, groups = score,
                  plot.points = FALSE,
                  auto.key = list(space = "right", title = "score"))
```

When these commands are executed, nothing gets plotted. In fact, `tp1` and `tp2` are now objects of class *"trellis"* that can, for instance, be summarized:

```
> class(tp2)
[1] "trellis"
> summary(tp1)
Call:
histogram(~gcsescore | factor(score), data = Chem97)

Number of observations:
factor(score)
```

```
   0    2    4    6    8   10
3688 3627 4619 5739 6668 6681
```

As noted above, the actual plots can be drawn by calling `print()`:

```
> print(tp1)
```

This may seem somewhat unintuitive, because `print()` normally produces text output in R, but it is necessary to take advantage of the automatic printing rule. The more natural

```
> plot(tp1)
```

has the same effect.

1.3.1 The missing Trellis display

Due to the automatic invocation of `print()`, lattice functions usually work as traditional graphics functions, where graphics output is generated when the user calls a function. Naturally, this similarity breaks down in contexts where automatic printing is suppressed. This happens, as we have seen, when the result of a lattice call is assigned to a variable. Unfortunately, it may also happen in other situations where the user may not be expecting it, for example, within `for()` or `while()` loops, or inside other functions. This includes the `source()` function, which is often used to execute an external R script, unless it is called with the `echo` argument set to `TRUE`. As with regular (non-graphics) R calls, the solution is to `print()` (or `plot()`) the result of the lattice call explicitly.

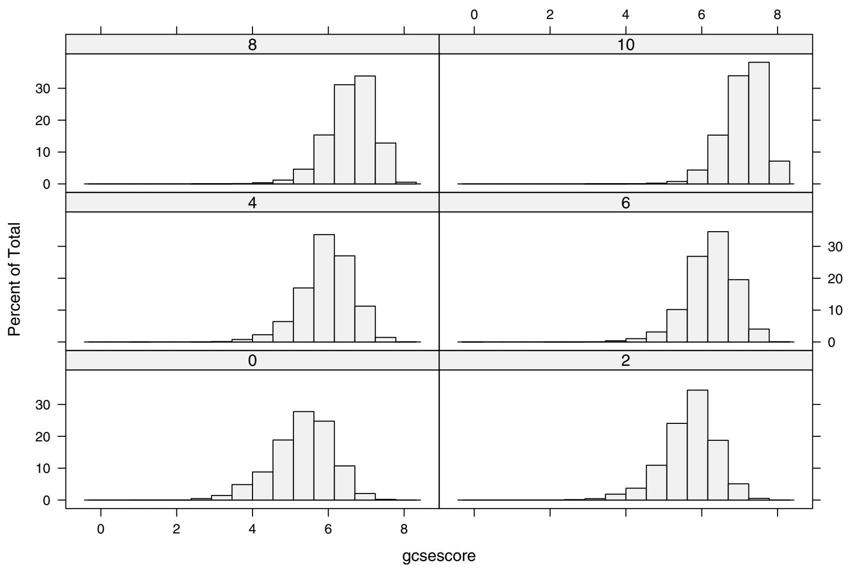
1.3.2 Arranging multiple Trellis plots

This object-based design has many useful implications, chief among them being the ability to arrange multiple lattice displays on a single page. Multipanel conditioning obviates the need for such usage to a large extent, but not entirely. For example, in Figure 1.4 we directly contrast the conditional histograms and the grouped density plots seen before. This is achieved by specifying the subregion to be occupied by a graphic on the fly when it is drawn, using optional arguments of the `plot()` method. Although this is one of the most common manipulations involving *"trellis"* objects explicitly, it is by no means the only one. A detailed discussion of *"trellis"* objects is given in Chapter 11.

1.4 Looking ahead

We have encountered two lattice functions in this chapter, `histogram()` and `densityplot()`. Each produces a particular type of statistical graphic, helpfully hinted at by its name. This sets the general trend: the lattice user interface principally consists of these and several other functions like these, each

```
> plot(tp1, split = c(1, 1, 1, 2))
> plot(tp2, split = c(1, 2, 1, 2), newpage = FALSE)
```



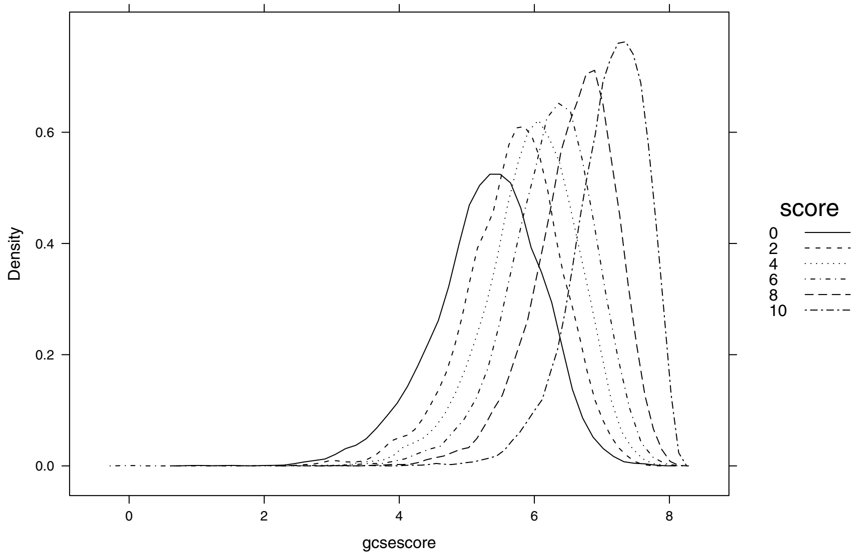


Figure 1.4. The conditional histogram and the grouped density plot of `gcsescore` by `score`, combined in a single figure. The comparison clearly illustrates the usefulness of superposition; the pattern of variance decreasing with mean that is obvious in the density plot is easy to miss in the histogram.

Function	**Default Display**
`histogram()`	Histogram
`densityplot()`	Kernel Density Plot
`qqmath()`	Theoretical Quantile Plot
`qq()`	Two-sample Quantile Plot
`stripplot()`	Stripchart (Comparative 1-D Scatter Plots)
`bwplot()`	Comparative Box-and-Whisker Plots
`dotplot()`	Cleveland Dot Plot
`barchart()`	Bar Plot
`xyplot()`	Scatter Plot
`splom()`	Scatter-Plot Matrix
`contourplot()`	Contour Plot of Surfaces
`levelplot()`	False Color Level Plot of Surfaces
`wireframe()`	Three-dimensional Perspective Plot of Surfaces
`cloud()`	Three-dimensional Scatter Plot
`parallel()`	Parallel Coordinates Plot

Table 1.1. High-level functions in the lattice package and their default displays.

intended to produce a particular type of graphic by default. The full list of high-level functions in lattice is given in Table 1.1. Chapters 3 through 6 focus on the capabilities of these high-level functions, describing each one in turn. The functions have much in common: they each have a formula interface that supports multipanel conditioning in a consistent manner, and respond to a number of common arguments. These common features, including the basics of multipanel conditioning, are described briefly in Chapter 2, and in further detail in Chapters 7 through 12. lattice is designed to be easily extensible using panel functions; some nontrivial examples are given in Chapter 13. Extensions can also be implemented as new high-level functions; Chapter 14 gives some examples and provides pointers for those who wish to create their own.