10

Data Manipulation and Related Topics

Now that we have had a chance to look at several types of lattice plots and ways to control their various elements, it is time to take another look at the big picture and introduce some new ideas. This chapter may be viewed as a continuation of Chapter 2; the topics covered are slightly more advanced, but generally apply to all lattice functions.

10.1 Nonstandard evaluation

Variables in the Trellis formula (along with those in `groups` and `subset`, if supplied) are evaluated in an optional data source specified as the `data` argument. This is usually a data frame, but could also be a list or environment. (Other types of data sources can be handled by writing new methods, as we see in Chapter 14.) When a term in the formula (or `groups` or `subset`) involves a variable not found in `data`, special scoping rules apply to determine its value; it is searched for in the environment of the formula, and if not found there, in the enclosing environment, and so on. In other words, the search does not start in the environment where the term is being evaluated, as one might expect. If no `data` argument is specified in a lattice call, the search for all variables starts in the environment of the formula. This behavior is similar to that in other formula-based modeling functions (e.g., `lm()`, `glm()`, etc.), and is commonly referred to as "standard nonstandard evaluation".

This is not an entirely academic issue. There are situations where this nonstandard scoping behavior is desirable, and others where it gives "unexpected" results. To get a sense of the issues involved, consider the following example where we revisit the choice of an optimal Box–Cox transformation for the `gcsescore` variable in the `Chem97` data (Figure 3.7). Instead of choosing the transformation analytically, we might simply try out several choices and visualize the results. We might do this by first defining a function implementing the Box–Cox transformation

```
> boxcox.trans <- function(x, lambda) {
      if (lambda == 0) log(x) else (x^lambda - 1) / lambda
  }
```

which is then used to create a multipage PDF file.

```
> data(Chem97, package = "mlmRev")
> trellis.device(pdf, file = "Chem97BoxCox.pdf",
                 width = 8, height = 6)
> for (p in seq(0, 3, by = 0.5)) {
      plot(qqmath(~boxcox.trans(gcsescore, p) | gender, data = Chem97,
                  groups = score, f.value = ppoints(100),
                  main = as.expression(substitute(lambda == v,
                                                  list(v = p)))))
  }
> dev.off()
```

In this example, the variable `p` in the formula is not visible in the `data` argument, and according to the nonstandard evaluation rules, it is searched for (and found) in the environment in which the formula was defined. This is the right thing to do in this case; we would not have wanted to use any other variable named `p` that might have been visible in the environment where the terms in the formula are actually evaluated. On the other hand, someone used to the standard lexical scoping behavior in R might think that the following is a reasonable alternative.

```
> form <- ~ boxcox.trans(gcsescore, p) | gender
> qqboxcox <- function(lambda) {
      for (p in lambda)
          plot(qqmath(form, data = Chem97,
                      groups = score, f.value = ppoints(100),
                      main = as.expression(substitute(lambda == v,
                                                      list(v = p)))))
  }
> qqboxcox(lambda = seq(0, 3, by = 0.5))
```

However, this will either fail because `p` is not found, or worse, use the wrong value of `p`. Of course, this is a rather artificial and perhaps not very convincing example. Most of the real problems due to nonstandard evaluation arise when trying to implement new wrapper functions with similar semantics, especially because the nonstandard evaluation rules also apply to the `groups` and `subset` arguments. One standard solution is outlined in the final example in Chapter 14.

10.2 The extended formula interface

We have already encountered the `Titanic` dataset in Chapter 2. To use the data in a lattice plot, it is convenient to coerce them into a data frame, as we do here for the subset of adults:

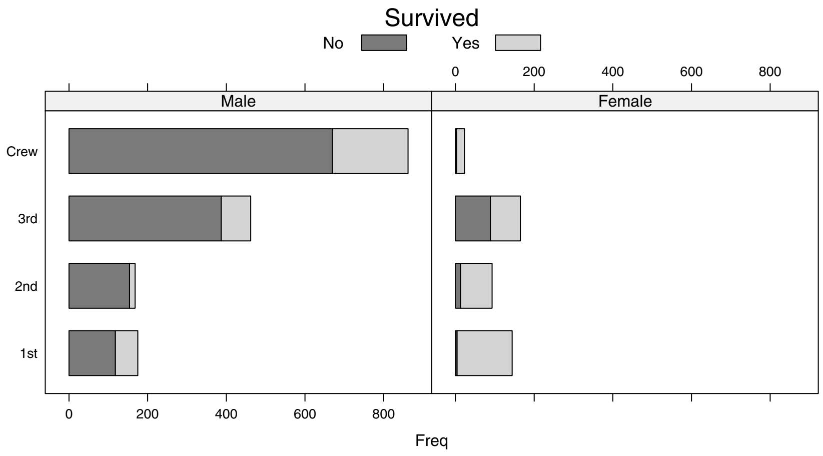

Figure 10.1. A bar chart showing the fate of adult passengers of the Titanic.

```
> Titanic1 <- as.data.frame(as.table(Titanic[, , "Adult" ,]))
> Titanic1
   Class    Sex Survived Freq
1    1st   Male       No  118
2    2nd   Male       No  154
3    3rd   Male       No  387
4   Crew   Male       No  670
5    1st Female       No    4
6    2nd Female       No   13
7    3rd Female       No   89
8   Crew Female       No    3
9    1st   Male      Yes   57
10   2nd   Male      Yes   14
11   3rd   Male      Yes   75
12  Crew   Male      Yes  192
13   1st Female      Yes  140
14   2nd Female      Yes   80
15   3rd Female      Yes   76
16  Crew Female      Yes   20
```

This form of the data is perfectly suited to our purposes. For example, Figure 10.1 can be produced from it by

```
> barchart(Class ~ Freq | Sex, Titanic1,
           groups = Survived, stack = TRUE,
           auto.key = list(title = "Survived", columns = 2))
```

Unfortunately, data may not always be as conveniently formatted. For example, these data could easily have been specified in the so-called "wide" format (as opposed to the "long" format above):

```
> Titanic2
  Class    Sex Dead Alive
1   1st   Male  118    57
2   2nd   Male  154    14
3   3rd   Male  387    75
4  Crew   Male  670   192
5   1st Female    4   140
6   2nd Female   13    80
7   3rd Female   89    76
8  Crew Female    3    20
```

This format is particularly common for longitudinal data, where multiple observations (e.g., over time) on a single experimental unit are often presented in one row rather than splitting them up over several rows (in which case covariates associated with the experimental units would have to be repeated). The formula interface described in Chapter 2 is not up to handling the seemingly simple task of reproducing Figure 10.1 from the data in the wide format.

The traditional solution is to transform the data into the long format before plotting. This can be accomplished using the `reshape()` function; in fact, our artificial example was created using

```
> Titanic2 <-
      reshape(Titanic1, direction = "wide", v.names = "Freq",
              idvar = c("Class", "Sex"), timevar = "Survived")
> names(Titanic2) <- c("Class", "Sex", "Dead", "Alive")
```

Unfortunately, `reshape()` is not the simplest function to use, and as this kind of usage is common enough, lattice provides a way to avoid calling `reshape()` by encoding the desired transformation within the formula. In particular, the part of the formula specifying primary variables (to the left of the conditioning symbol) can contain multiple terms separated by a `+` symbol, in which case they are treated as columns in the wide format that are to be concatenated to form a single column in the long format. Figure 10.2 is produced by

```
> barchart(Class ~ Dead + Alive | Sex, Titanic2, stack = TRUE,
           auto.key = list(columns = 2))
```

Notice that the new factor implicitly created (to indicate from which column in the wide format a row in the long format came) has been used for grouping without any explicit specification of the `groups` argument. This is the default behavior for high-level functions in which grouped displays make sense. One may instead want to use the new factor as a conditioning variable; this can be done by specifying `outer = TRUE` in the call. In either case, the `subscripts` argument, if used in a custom panel function, refers to the implicitly reshaped data.

An alternative interpretation of such formulae that avoids the concept of reshaping is as follows: the formula `y1 + y2 ~ x | a` should be taken to mean that the user wishes to plot both `y1 ~ x | a` and `y2 ~ x | a`, with `outer` determining whether the plots are to use the same or separate panels.

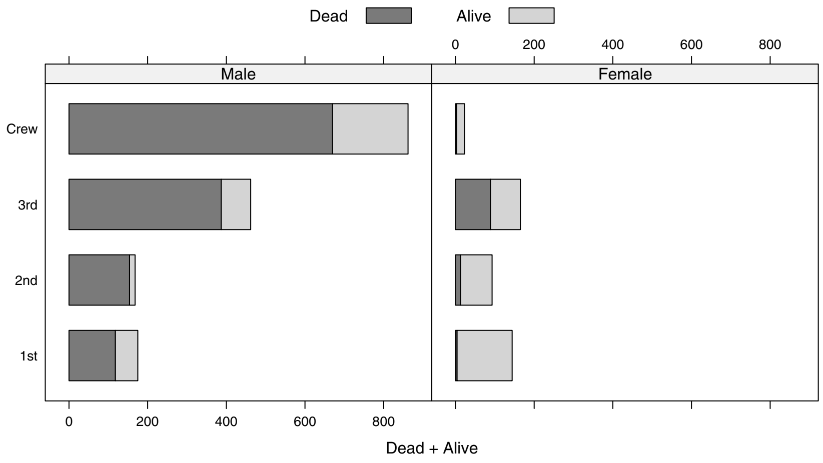

Figure 10.2. An alternative formulation of Figure 10.1, using data in the wide format. The plots are identical, except for the legend and the x-axis label.

This behavior is distributive, in the sense that the formula `y1 + y2 ~ x1 + x2` will cause all four combinations (`y1 ~ x1`, `y1 ~ x2`, `y2 ~ x1`, and `y2 ~ x2`) to be displayed. To interpret `y1 + y2` as a sum in the formula, one can use `I(y1 + y2)`, which suppresses the special interpretation of `+`.

For another example where the use of the extended formula interface arises naturally, consider the `Gcsemv` dataset (Rasbash et al., 2000) in the mlmRev package.

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

The data record the GCSE exam scores of 1905 students in England on a science subject. The scores for two components are recorded: written paper and course work. The scores are paired, so it is natural to consider a scatter plot of the written and coursework scores conditioning on `gender`. Figure 10.3 is produced by

```
> xyplot(written ~ course | gender, data = Gcsemv,
         type = c("g", "p", "smooth"),
         xlab = "Coursework score", ylab = "Written exam score",
         panel = function(x, y, ...) {
             panel.xyplot(x, y, ...)
             panel.rug(x = x[is.na(y)], y = y[is.na(x)])
         })
```

where we use the predefined panel function `panel.rug()`[1] to encode the scores for cases where one component is missing (these would otherwise have been omitted from the plot). This plot clearly suggests an association between the

[1] See Chapter 13 for a full list of predefined panel functions.

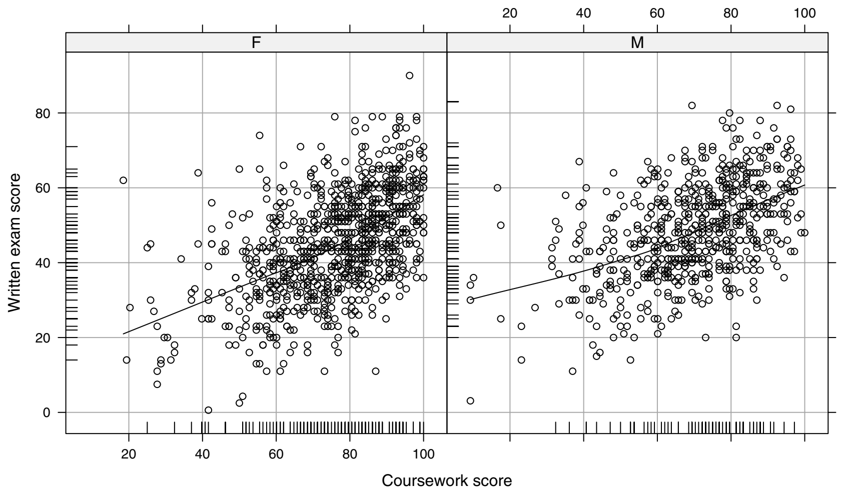

Figure 10.3. A scatter plot of coursework and written exam scores on a science subject, conditioned on gender. Scores missing in one variable are indicated by “rugs”.

two scores. In the next plot, we ignore the pairing of the scores and look at their marginal distributions using a Q–Q plot. Figure 10.4 is produced by

```
> qqmath(~ written + course, Gcsemv, type = c("p", "g"),
         outer = TRUE, groups = gender, auto.key = list(columns = 2),
         f.value = ppoints(200), ylab = "Score")
```

This plot emphasizes two facts about the marginal distributions: boys tend to do slightly better than girls in the written exam whereas the opposite is true for coursework, and although the written scores fit a normal distribution almost perfectly, the coursework scores do not. In fact, the distributions of coursework scores have a positive probability mass at 100 (one might say that the “true” scores have been right censored), more so for girls than boys. Neither of these facts are unexpected, but they are not as obvious in the previous scatter plot.

10.3 Combining data sources with `make.groups()`

By itself, the formula interface is not flexible enough for all situations, and one often needs to manipulate the data before using them in a `lattice` call. One common scenario is when datasets of different lengths need to be combined. All terms in the Trellis formula (even in the extended form) should have the same length after evaluation. This restriction is naturally enforced when `data` is a data frame, but there is no explicit check for other data sources. This can sometimes be an issue if one is careless, especially with “univariate” formu-

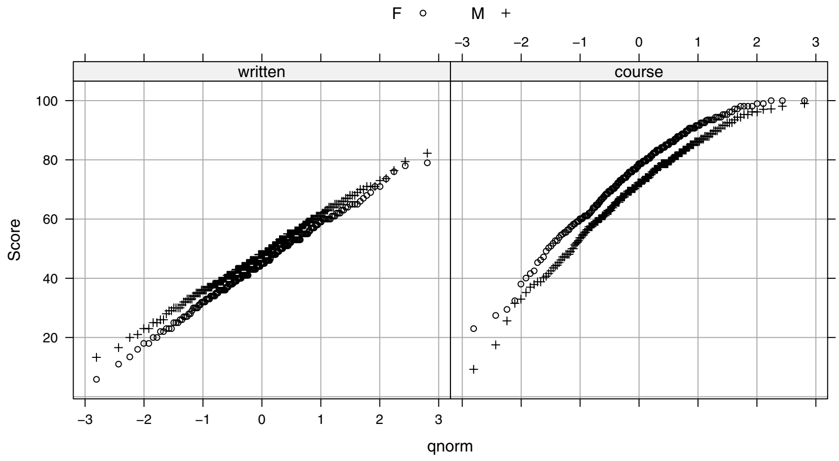

Figure 10.4. Normal quantile plots of written exam and coursework scores, grouped by gender. The distributions of written exam scores are close to normal, with males doing slightly better. The distributions of coursework scores are skewed, with several full scores for females (note that only a subset of quantiles has been plotted), who do considerably better. The comparison is visually more striking when color is used to distinguish between the groups.

lae as used in Q–Q plots and histograms. To make this point, consider this somewhat artificial example: Among continuous probability distributions, the exponential distribution is unique in having the property (often referred to as the memoryless property) that left truncation is equivalent to an additive shift in the induced distribution. To demonstrate this using a Q–Q plot, we generate truncated and untruncated observations from the standard exponential distribution.

```
> x1 <- rexp(2000)
> x1 <- x1[x1 > 1]
> x2 <- rexp(1000)
```

In view of the preceding discussion, one might be tempted to try something along the lines of

```
> qqmath(~ x1 + x2, distribution = qexp)
```

to create a grouped theoretical Q–Q plot, but this produces the wrong output because `x1` and `x2` are not of equal length. The correct approach is to combine the vectors and form a suitable grouping variable, as in

```
> qqmath( ~ c(x1, x2), distribution = qexp,
         groups = rep(c("x1", "x2"), c(length(x1), length(x2))))
```

This is of course tedious, even more so when there are more objects to combine. A utility function designed for such situations is `make.groups()`, which

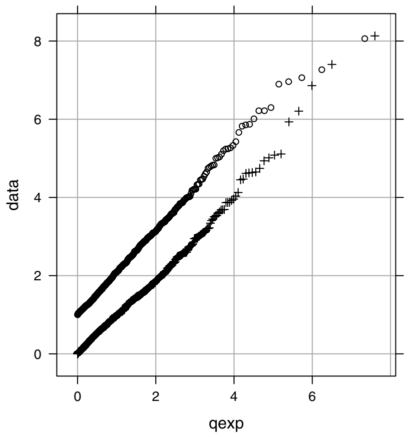

Figure 10.5. Theoretical quantile plot comparing the untruncated and truncated (at 1) exponential distributions. The plot illustrates the use of `make.groups()` as discussed in the text.

combines several vectors, possibly of different lengths, into a single data frame with two columns: one (`data`) concatenating all its arguments, the other (`which`) indicating from which vector an observation came. For example, we have

```
> str(make.groups(x1, x2))
'data.frame':   1772 obs. of  2 variables:
 $ data : num  2.31 2.35 2.49 1.51 ...
 $ which: Factor w/ 2 levels "x1","x2": 1 1 1 1 1 1 1 1 ...
```

We can thus produce Figure 10.5 with

```
> qqmath(~ data, make.groups(x1, x2), groups = which,
         distribution = qexp, aspect = "iso", type = c("p", "g"))
```

Multiple data frames with differing number of rows can also be combined using `make.groups()`, provided they have conformable columns. As an example, consider the `beavers` dataset (Reynolds, 1994; Venables and Ripley, 2002), which actually consists of two data frames `beaver1` and `beaver2`, recording body temperature of two beavers in north-central Wisconsin every ten minutes over a period of several hours.

```
> str(beaver1)
'data.frame':   114 obs. of  4 variables:
 $ day  : num  346 346 346 346 346 346 346 346 ...
 $ time : num  840 850 900 910 920 930 940 950 ...
 $ temp : num  36.3 36.3 36.4 36.4 ...
 $ activ: num  0 0 0 0 0 0 0 0 ...
> str(beaver2)
```

```
'data.frame':   100 obs. of  4 variables:
 $ day  : num  307 307 307 307 307 307 307 307 ...
 $ time : num  930 940 950 1000 1010 1020 1030 1040 ...
 $ temp : num  36.6 36.7 36.9 37.1 ...
 $ activ: num  0 0 0 0 0 0 0 0 ...
```

We can combine these in a single data frame using

```
> beavers <- make.groups(beaver1, beaver2)
> str(beavers)
'data.frame':   214 obs. of  5 variables:
 $ day  : num  346 346 346 346 346 346 346 346 ...
 $ time : num  840 850 900 910 920 930 940 950 ...
 $ temp : num  36.3 36.3 36.4 36.4 ...
 $ activ: num  0 0 0 0 0 0 0 0 ...
 $ which: Factor w/ 2 levels "beaver1","beaver2": 1 1 1 1 1 1 1 1 ...
```

The time of each observation is recorded in a somewhat nonstandard manner. To use them in a plot, one option is to convert them into hours past an arbitrary baseline using

```
> beavers$hour <-
      with(beavers, time %/% 100 + 24*(day - 307) + (time %% 100)/60)
```

The range of this new variable is very different for the two beavers (the two sets of measurements were taken more than a month apart), so plotting them on a common axis does not make sense. We could of course measure hours from different baselines for each beaver, but another alternative is to allow different limits using

```
> xyplot(temp ~ hour | which, data = beavers, groups = activ,
         auto.key = list(text = c("inactive", "active"), columns = 2),
         xlab = "Time (hours)", ylab = "Body Temperature (C)",
         scales = list(x = list(relation = "sliced")))
```

The result is shown in Figure 10.6. This is a natural use of `"sliced"` scales, as we want differences in time to be comparable across panels, even though the absolute values have no meaningful interpretation.

10.4 Subsetting

As with other formula-based interfaces in R (such as `lm()` and `glm()`), one can supply a `subset` argument to choose a subset of rows to use in the display. If specified, it should be an expression, possibly involving variables in `data`, that evaluates to a logical vector. The result should have the same length as the number of rows in the data, and is recycled if not. For example, the graphic in Figure 10.1 (which uses only the subset of adults) could have been obtained directly from the full `Titanic` data by

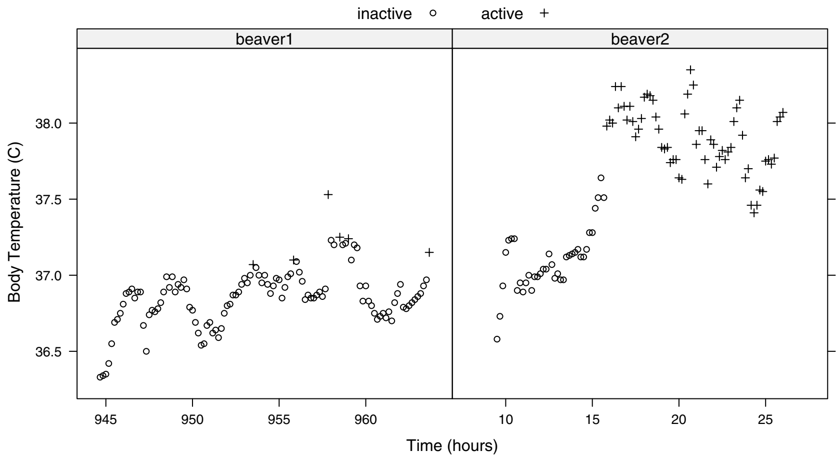

Figure 10.6. Body temperature of two beavers (over time) in north-central Wisconsin. The plotting symbols indicate periods of outside activity, which clearly affects body temperature.

```
> barchart(Class ~ Freq | Sex, as.data.frame(Titanic),
           subset = (Age == "Adult"), groups = Survived, stack = TRUE,
           auto.key = list(title = "Survived", columns = 2))
```

Subsetting becomes more important for larger datasets. To illustrate this, let us consider the `USAge.df` dataset in the latticeExtra package, which records estimated population[2] of the United States by age and sex for the years 1900 through 1979.

```
> data(USAge.df, package = "latticeExtra")
> head(USAge.df)
  Age  Sex Year Population
1   0 Male 1900      0.919
2   1 Male 1900      0.928
3   2 Male 1900      0.932
4   3 Male 1900      0.932
5   4 Male 1900      0.928
6   5 Male 1900      0.921
```

Figure 10.7 plots the population distribution for every tenth year starting with 1905:

```
> xyplot(Population ~ Age | factor(Year), USAge.df,
         groups = Sex, type = c("l", "g"),
         auto.key = list(points = FALSE, lines = TRUE, columns = 2),
         aspect = "xy", ylab = "Population (millions)",
         subset = Year %in% seq(1905, 1975, by = 10))
```

[2] Source: U.S. Census Bureau, `http://www.census.gov`.

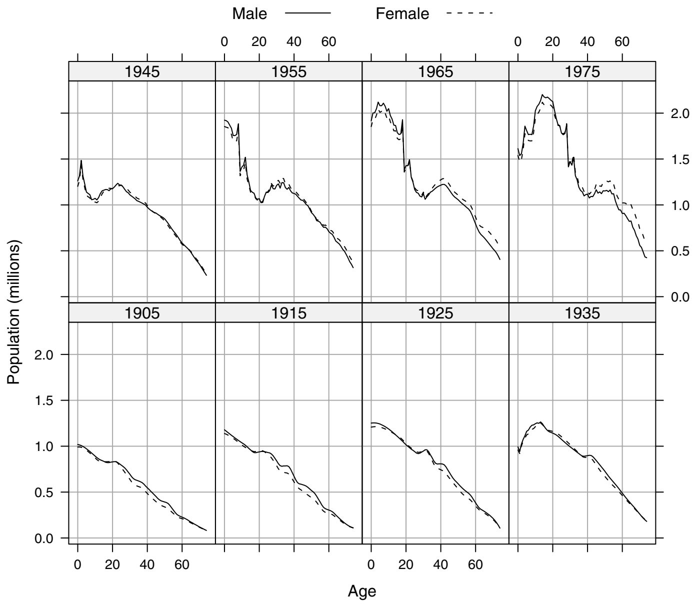

Figure 10.7. U.S. population distribution by gender, every ten years from 1905 through 1975.

The "baby boom" phenomenon of the late 1940s and 1950s is clearly apparent from the plot. It is clearer in the next representation, where each panel represents a specific age, and plots the population for that age over the years. Figure 10.8 is produced by

```
> xyplot(Population ~ Year | factor(Age), USAge.df,
         groups = Sex, type = "l", strip = FALSE, strip.left = TRUE,
         layout = c(1, 3), ylab = "Population (millions)",
         auto.key = list(lines = TRUE, points = FALSE, columns = 2),
         subset = Age %in% c(0, 10, 20))
```

In particular, the panel for age 0 represents the number of births (ignoring immigration, which is less important here than in the older age groups). A closer look at the panel for 20-year-olds shows an intriguing dip in the male population around 1918. To investigate this further, the next plot follows the population distribution by cohort; Figure 10.9 conditions on the year of birth, and is produced by

```
> xyplot(Population ~ Year | factor(Year - Age), USAge.df,
         groups = Sex, subset = (Year - Age) %in% 1894:1905,
         type = c("g", "l"), ylab = "Population (millions)",
         auto.key = list(lines = TRUE, points = FALSE, columns = 2))
```

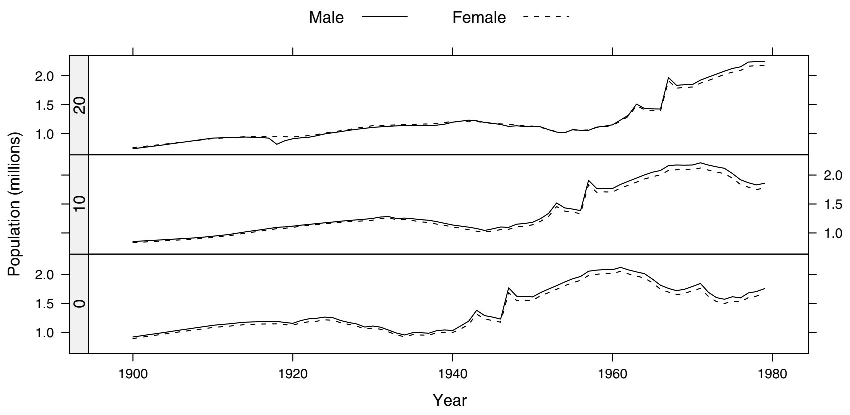

Figure 10.8. U.S. population by age over the years. The bottom panel gives the (approximate) number of births each year, clearly showing the baby boom after World War II. The top panel, which gives the population of 20-year-olds, shows a temporary drop in the male population around 1918.

Unlike in the previous plots, no individual is counted in more than one panel. The signs of some major event in or around 1918 is clear, and a closer look suggests that its impact on population varies by age and sex. The most natural explanation is that the fluctuation is related to the United States joining World War I in 1917; however, there is no similar fluctuation for World War II. As it turns out, armed forces stationed overseas were excluded from the population estimates for the years 1900-1939, but not for subsequent years.

10.4.1 Dropping of factor levels

A subtle point that is particularly relevant when using the `subset` argument is the rule governing dropping of levels in factors. By default, levels that are unused (i.e., have no corresponding data points) after subsetting are omitted from the plot. This behavior can be changed by the `drop.unused.levels` argument separately for conditioning variables and panel variables. The default behavior is usually reasonable, but the ability to override it is often helpful in obtaining a more useful layout. Note, however, that levels of a grouping variable are never dropped automatically. This is because unlike variables in the formula, subsetting of `groups` is done inside panel functions, and dropping levels in this case may inadvertently lead to inconsistency across panels or meaningless legends. A possible workaround is described in Chapter 9 in the context of the `auto.key` argument.

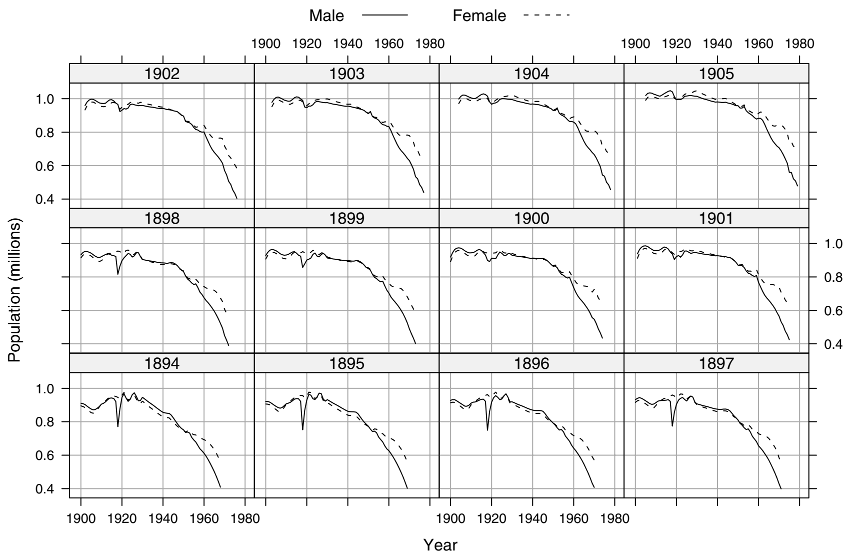

Figure 10.9. U.S. population by cohort (birth year), showing the effect of age and sex on the temporary drop in population in 1918. The use of broken lines for females leads to unequal visual emphasis on the two groups that is completely artificial; compare with the color version of this figure, which is included among the color plates.

10.5 Shingles and related utilities

We have briefly encountered shingles previously in Chapter 2. In this section, we take a closer look at the facilities available to construct and manipulate shingles. As an example, we use the `quakes` dataset, and look at how the number of stations reporting an earthquake is related to its magnitude. Figure 10.10 is produced by

```
> xyplot(stations ~ mag, quakes, jitter.x = TRUE,
         type = c("p", "smooth"),
         xlab = "Magnitude (Richter)",
         ylab = "Number of stations reporting")
```

Subject to certain assumptions, we might expect the counts to have a Poisson distribution, with mean related to earthquake magnitude. The documentation of the dataset notes that there are no quakes with magnitude less than 4.0 on the Richter scale, but a closer look at Figure 10.10 also reveals that there are none with less than ten reporting stations. This truncation makes it harder to decide whether the the expected count is a linear function of the magnitude, although the shape of the LOESS smooth for magnitude greater than 4.5 supports that conjecture. Another implication of the Poisson model is that the variance of the counts increases with the mean. We use shingles to investigate

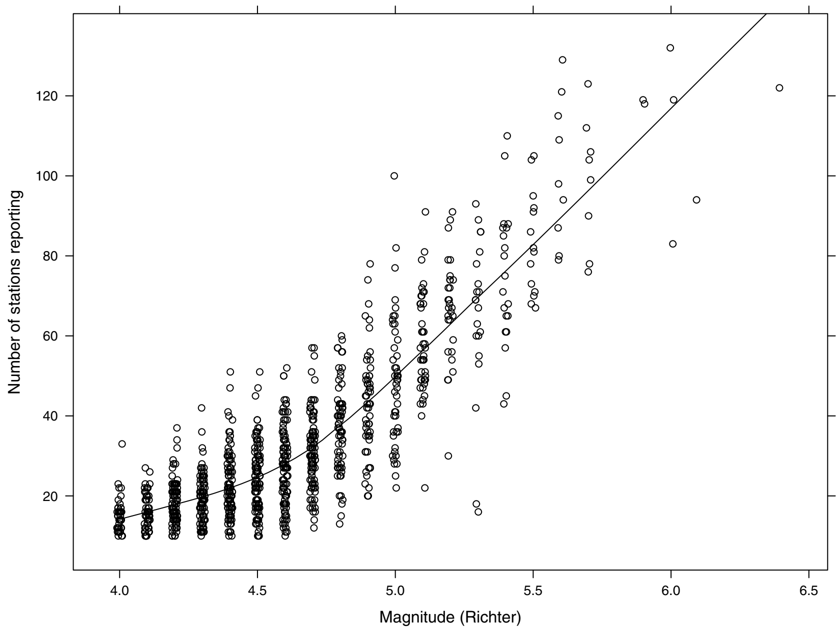

Figure 10.10. Number of stations recording earthquakes of various magnitudes. Earthquakes with less than ten stations reporting are clearly omitted from the data, which possibly explains the curvature in the LOESS smooth.

whether this seems to be true.[3] First, we construct a shingle from the numeric `mag` variable using the `equal.count()` function.

```
> quakes$Mag <- equal.count(quakes$mag, number = 10, overlap = 0.2)
```

This creates a shingle with ten levels, each represented by a numeric interval. The endpoints of the intervals are determined automatically (based on the data) so that roughly the same number of observations falls in each. The `overlap` argument determines the fraction of overlap between successive levels; in this case, 20% of the data in each interval should also belong to the next. The `overlap` can be negative, in which case there will be gaps in the coverage. It is also possible to create shingles with the intervals explicitly specified, using the `shingle()` function, as we soon show. The resulting levels and the corresponding frequencies can be inspected by summarizing the shingle:

```
> summary(quakes$Mag)
Intervals:
   min  max count
1 3.95 4.25   191
2 4.05 4.35   230
```

[3] This is by no means the only way to do so.

```
3  4.25 4.45   186
4  4.35 4.55   208
5  4.45 4.65   208
6  4.55 4.75   199
7  4.65 4.85   163
8  4.75 5.05   166
9  4.85 5.25   173
10 5.05 6.45   151

Overlap between adjacent intervals:
[1] 145  85 101 107 101  98  65 101  72
```

The nominal goals of having an equal number of observations in each level and an overlap of 20% between successive intervals have not quite been met, but this is a consequence of heavy rounding of the magnitudes, with only 22 unique values. A character representation of the levels, useful for annotation, is produced by

```
> as.character(levels(quakes$Mag))
 [1] "[ 3.95, 4.25 ]" "[ 4.05, 4.35 ]" "[ 4.25, 4.45 ]"
 [4] "[ 4.35, 4.55 ]" "[ 4.45, 4.65 ]" "[ 4.55, 4.75 ]"
 [7] "[ 4.65, 4.85 ]" "[ 4.75, 5.05 ]" "[ 4.85, 5.25 ]"
[10] "[ 5.05, 6.45 ]"
```

A visual representation of the shingle can be produced using the `plot()` method[4] for shingles. This actually creates a *"trellis"* object, which we store in the variable `ps.mag` for now instead of plotting it.

```
> ps.mag <- plot(quakes$Mag, ylab = "Level",
                 xlab = "Magnitude (Richter)")
```

Next, we create another *"trellis"* object representing a box-and-whisker plot with `stations` on the y-axis and the newly created shingle on the x-axis.

```
> bwp.quakes <-
      bwplot(stations ~ Mag, quakes, xlab = "Magnitude",
             ylab = "Number of stations reporting")
```

Finally, we plot these *"trellis"* objects together to produce Figure 10.11.

```
> plot(bwp.quakes, position = c(0, 0, 1, 0.65))
> plot(ps.mag, position = c(0, 0.65, 1, 1), newpage = FALSE)
```

Without the plot of the shingle, we would not be able to associate a level of the shingle to the numeric interval it represents. An alternative is to manually annotate the shingle levels, as in Figure 10.12, which is produced by

```
> bwplot(sqrt(stations) ~ Mag, quakes,
         scales =
         list(x = list(limits = as.character(levels(quakes$Mag)),
                       rot = 60)),
```

[4] See `?plot.shingle` for details.

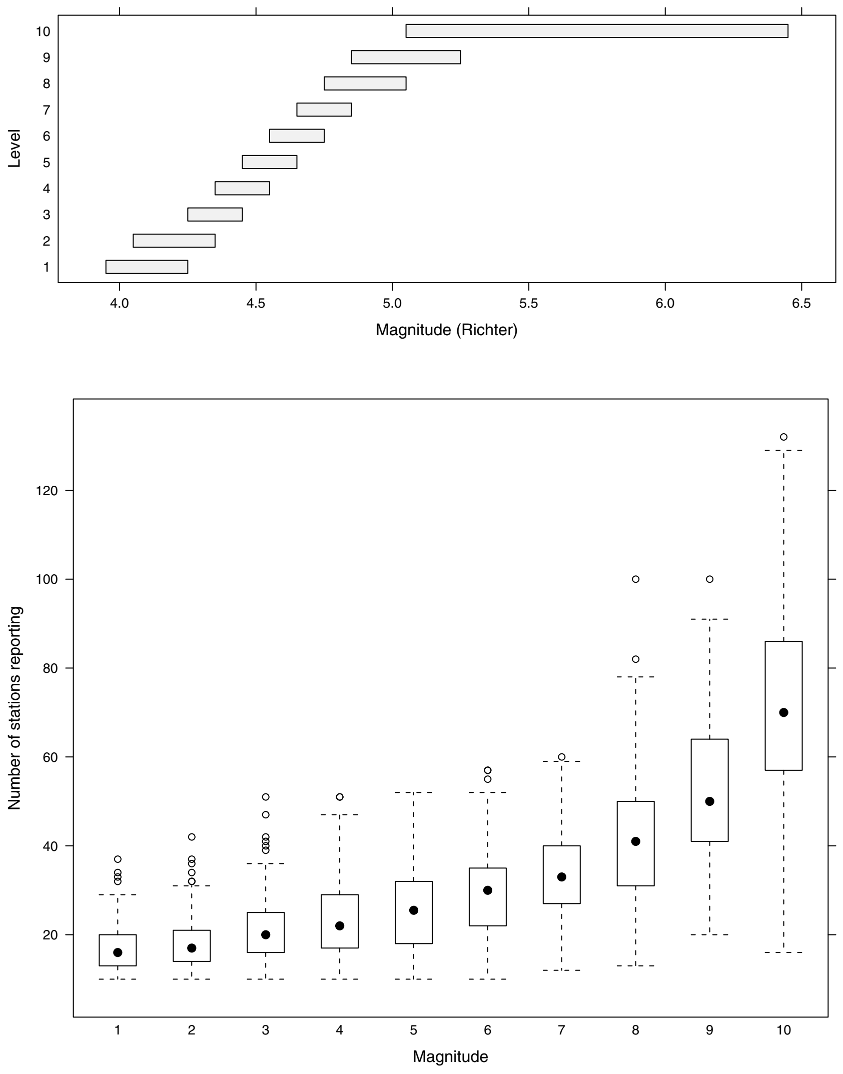

Figure 10.11. A box-and-whisker plot of the number of stations recording earthquakes, with a shingle representing the earthquake magnitudes. A plot of the shingle at the top gives the association between levels of the shingle and the corresponding ranges of magnitude on the Richter scale.

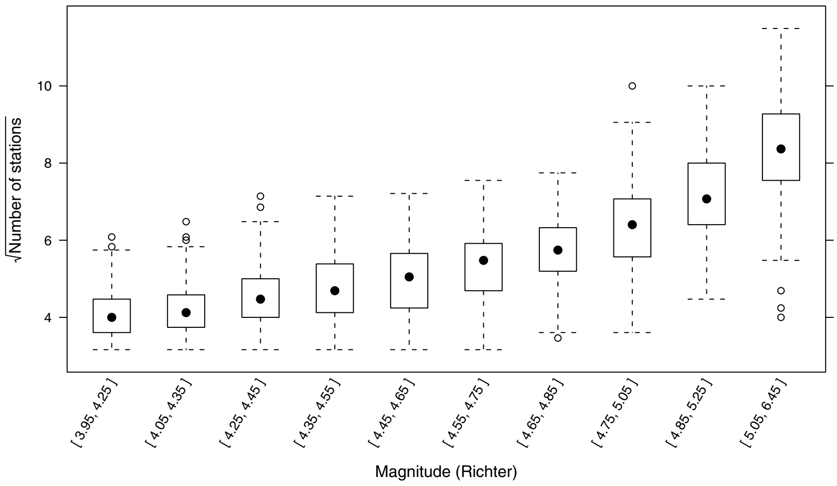

Figure 10.12. A variant of Figure 10.11. The y-axis now plots the square root of the number of stations reporting, and the range of each level of the shingle is given in the form of axis labels.

```
         xlab = "Magnitude (Richter)",
         ylab = expression(sqrt("Number of stations")))
```

where we additionally plot the number of reporting stations on a square root scale. The square root transformation is a standard variance stabilizing transformation for the Poisson distribution (Bartlett, 1936), and does seem to work reasonably well, given the omission of quakes reported by less than ten stations.

It is more common to use shingles as conditioning variables. In that case, the interval defining a level of the shingle is indicated relative to its full range by shading the strip region. If the exact numeric values of the interval are desired, one can add a plot of the shingle as in Figure 10.11. Another option is to print the numeric range inside each strip using a suitable strip function, as in the following call which produces Figure 10.13.

```
> qqmath(~ sqrt(stations) | Mag, quakes,
         type = c("p", "g"), pch = ".", cex = 3,
         prepanel = prepanel.qqmathline, aspect = "xy",
         strip = strip.custom(strip.levels = TRUE,
                              strip.names = FALSE),
         xlab = "Standard normal quantiles",
         ylab = expression(sqrt("Number of stations")))
```

The `strip.custom()` function used in this example is explained later in this chapter.

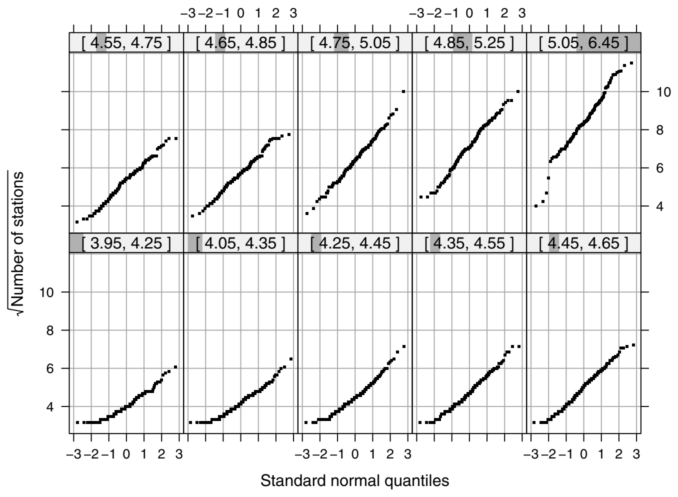

Figure 10.13. A normal quantile plot of the (square root of the) number of stations reporting earthquakes. The conditioning variable is a shingle, with numerical range shown in the strips. The truncation in the number of stations can be seen in the first few panels.

10.5.1 Coercion to factors and shingles

There are certain situations where lattice expects to find a "categorical variable" (i.e., either a factor or a shingle). This most obviously applies to conditioning variables; numeric variables are coerced to be shingles, and character variables are turned into factors. This coercion rule also applies in `bwplot()`, as well as a few other high-level functions, such as `stripplot()` and `barchart()`, which expect one of the axes to represent a categorical variable. If numeric variables are given for both axes, the choice of which one to coerce depends on the value of the `horizontal` argument.

This behavior can be frustrating, because there are occasions where we want a numeric variable to be used as a categorical variable in the display, yet retain the numeric scale for spacing and axis annotation. For example, we might want a plot with the same structure as Figure 10.10, but with the jittered point clouds for each value of `mag` replaced by a box-and-whisker plot. Unfortunately, attempting to do so with `bwplot()` will fail; the display produced by

```
> xyplot(stations ~ mag, quakes,
         panel = panel.bwplot, horizontal = FALSE)
```

will represent `mag` as a shingle with unique values equally spaced along the x-axis, and we will lose information about gaps in their values. One solution is to create a factor or shingle explicitly with empty levels. A simpler option

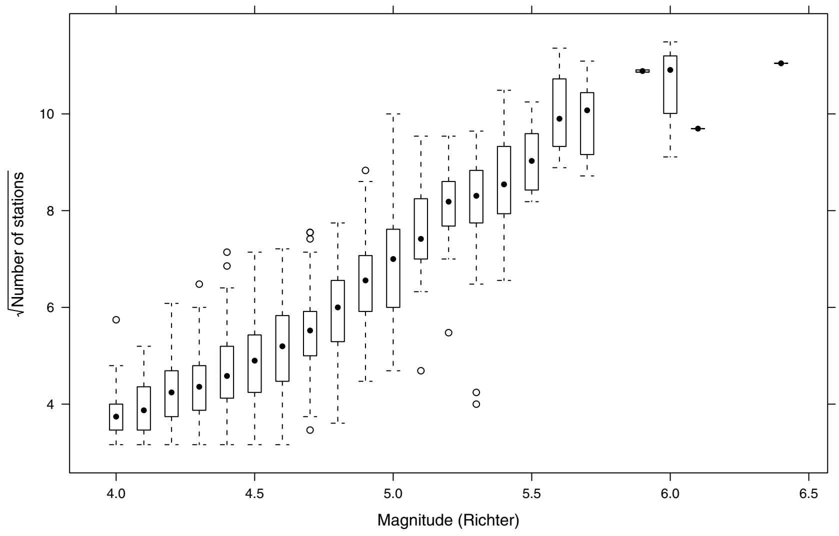

Figure 10.14. Box-and-whisker plots of number of reporting stations by magnitude of earthquakes. This time, the x variable is not a shingle, but the actual magnitude.

is not to use `bwplot()` at all, but instead use `xyplot()`, and borrow the panel function `panel.bwplot()`. Figure 10.14 is produced by

```
> xyplot(sqrt(stations) ~ mag, quakes, cex = 0.6,
         panel = panel.bwplot, horizontal = FALSE, box.ratio = 0.05,
         xlab = "Magnitude (Richter)",
         ylab = expression(sqrt("Number of stations")))
```

We need to tweak the `box.ratio` argument to account for the fact that successive boxes are not as far away from each other as `panel.bwplot()` expects.

10.5.2 Using shingles for axis breaks

Although shingles are commonly used for conditioning, they can be put to other interesting uses as well. In particular, a numeric variable can be used both in its original form (as a primary variable) and as a shingle (as a conditioning variable) in conjunction with the `relation` specification to create (possibly data-driven) scale breaks. As an example, consider the population density in the 50 U.S. states, based on population estimates from 1975:

```
> state.density <-
      data.frame(name = state.name,
                 area = state.x77[, "Area"],
                 population = state.x77[, "Population"],
                 region = state.region)
> state.density$density <- with(state.density, population / area)
```

We can produce a Cleveland dot plot of the raw densities using

```
> dotplot(reorder(name, density) ~ density, state.density,
          xlab = "Population Density (thousands per square mile)")
```

producing Figure 10.15. The plot is dominated by a few states with very high density, making it difficult to assess the variability among the remaining states. This kind of problem is usually alleviated by taking a logarithmic transformation, as we do later in Figures 10.19 through 10.21. However, another option is to create a break in the x-axis. There is achieved by creating a shingle, using the `shingle()` constructor, with suitable intervals specified explicitly.

```
> state.density$Density <-
      shingle(state.density$density,
              intervals = rbind(c(0, 0.2),
                                c(0.2, 1)))
```

This shingle can now be used as a conditioning variable to separate the states into two panels. Figure 10.16 is created using

```
> dotplot(reorder(name, density) ~ density | Density, state.density,
          strip = FALSE, layout = c(2, 1), levels.fos = 1:50,
          scales = list(x = "free"), between = list(x = 0.5),
          xlab = "Population Density (thousands per square mile)",
          par.settings = list(layout.widths = list(panel = c(2, 1))))
```

where the x-axis is allowed to be different for the two panels, and additionally the panel with more states is further emphasized by making it wider.

10.5.3 Cut-and-stack plots

Another use of shingles that is similar in spirit is to create so-called "cut-and-stack" plots (Cleveland, 1993). Time-series data are often best viewed with a low aspect ratio because local features are usually of more interest than overall trends. A suitable aspect ratio can usually be determined automatically using the 45° banking rule (`aspect = "xy"`), but this generally results in a short wide plot that does not make use of available vertical space. An easy way to remedy this is to divide up (cut) the time range into several smaller parts and stack them on top of each other. Shingles are ideal for defining the cuts because they allow overlaps, providing explicit continuity across panels. In fact, if we use the `equal.count()` function to create the shingle, all we need to specify is the number of cuts and the amount of overlap, as we did in Figure 8.2. We can wrap this procedure in a simple function:

```
> cutAndStack <-
      function(x, number = 6, overlap = 0.1, type = "l",
               xlab = "Time", ylab = deparse(substitute(x)), ...) {
      time <- if (is.ts(x)) time(x) else seq_along(x)
      Time <- equal.count(as.numeric(time),
                          number = number, overlap = overlap)
```

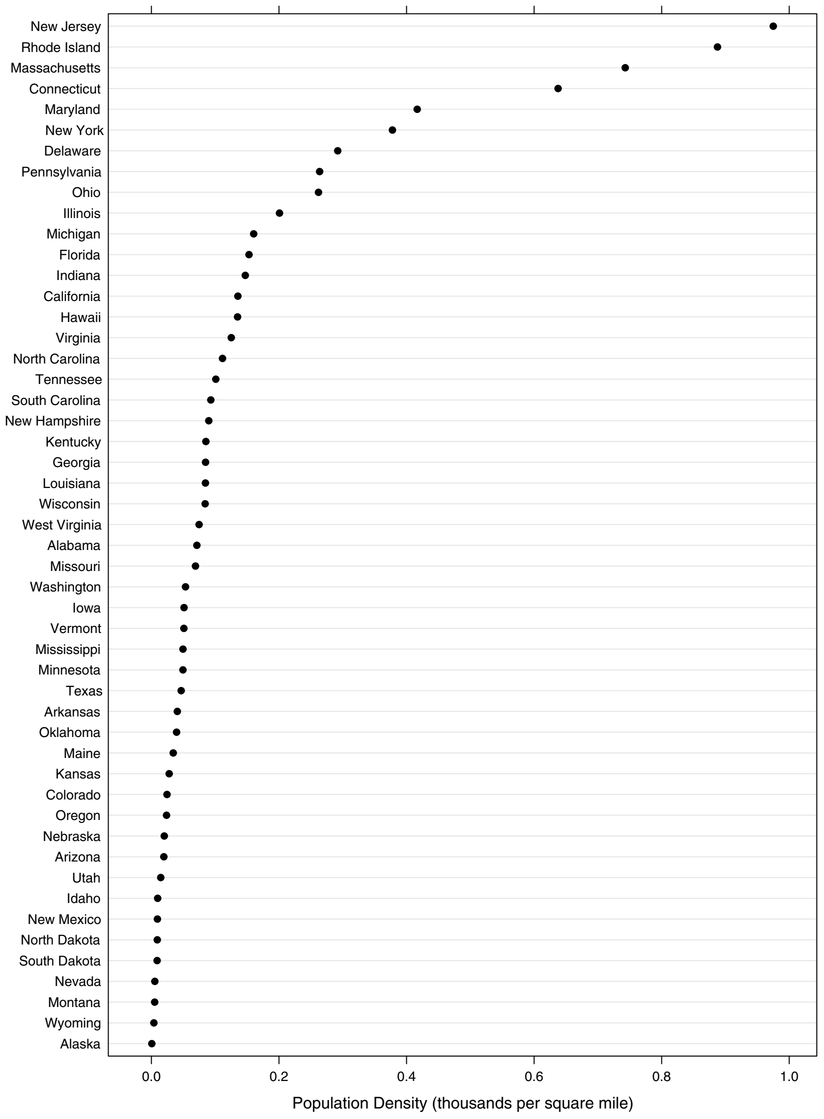

Figure 10.15. Estimated population density in U.S. states, 1975. A few extreme values dominate the plot.

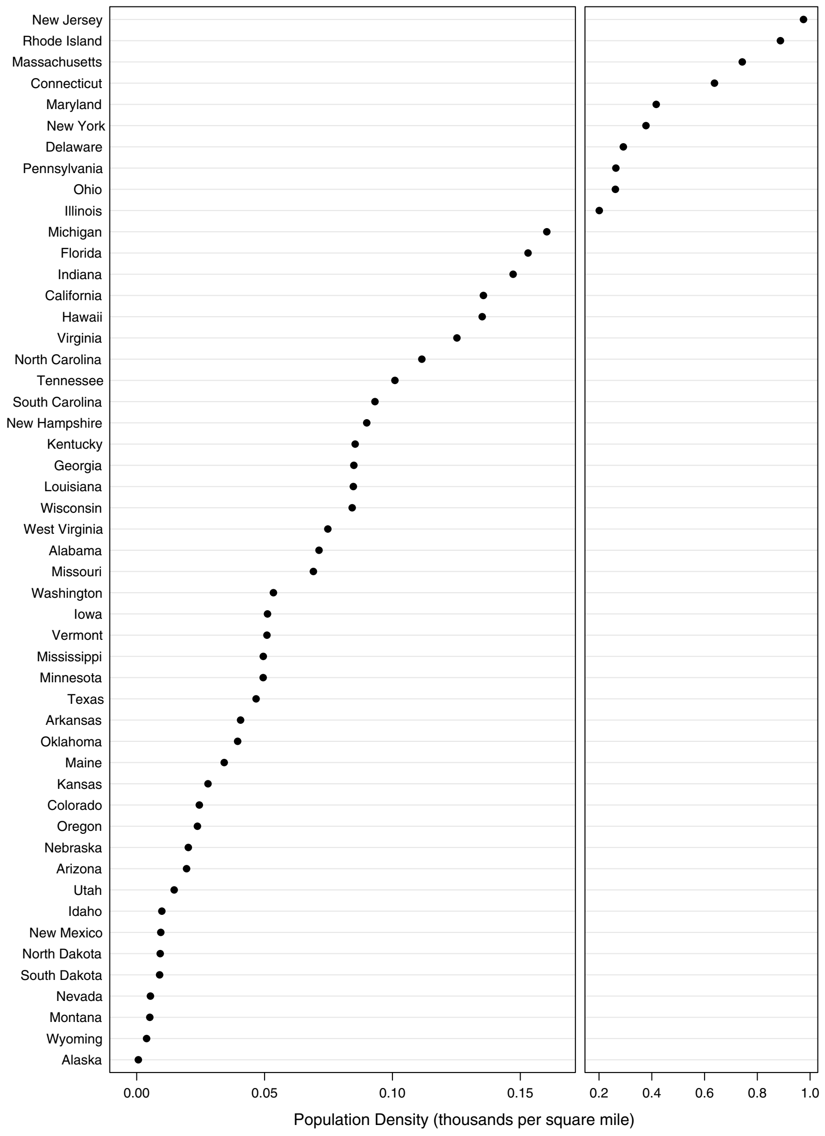

Figure 10.16. Estimated population density in U.S. states, with a break in the x-axis.

```
    xyplot(as.numeric(x) ~ time | Time,
           type = type, xlab = xlab, ylab = ylab,
           default.scales = list(x = list(relation = "free"),
                                 y = list(relation = "free")),
           ...)
}
```

We can then use this to create a cut-and-stack plot of a time-series object (class *"ts"*) or any other numeric vector. Figure 10.17 is produced by

```
> cutAndStack(EuStockMarkets[, "DAX"], aspect = "xy",
              scales = list(x = list(draw = FALSE),
                            y = list(rot = 0)))
```

An alternative approach is presented in Chapter 14.

10.6 Ordering levels of categorical variables

Unlike ordinal categorical variables and numeric variables (and by extension shingles), levels of nominal variables have no intrinsic order. An extremely important, but rarely appreciated fact is that the visual order of these levels in a display has considerable impact on how we perceive the information encoded in the display. By default, when R creates factors from character strings, it defines the levels in alphabetical order (this can be changed using the `levels` argument to the `factor()` constructor), and this order is retained in lattice plots. In most cases, reordering the levels based on the data, rather than keeping the original arbitrary order, leads to more informative plots.

This fact plays a subtle but important role in the well-known barley dot plot shown in Figure 2.6. In this plot, the levels of variety, site, and year were all ordered so that the median yield within level went from lowest to highest. We reproduce this plot in Figure 10.18 alongside a slightly different version where the levels of site and variety are ordered alphabetically. Although the switch in direction at Morris, the primary message from the plot, is clear in both plots, the one on the right makes it easier to convince ourselves that the likely culprit is simply a mislabeling of the year for that one site, and no more elaborate explanation is required.

Reordering levels of a factor with respect to the values of another variable is most easily done using the `reorder()` function.[5] We have already used `reorder()` in Chapter 4 to produce Figure 4.7, and earlier in this chapter when looking at dot plots of population densities in the United States. Continuing with the latter example, we can create a dot plot with log densities on the x-axis using

[5] `reorder()` is a generic function, and documentation for the method we are interested in can be accessed using `?reorder.factor`.

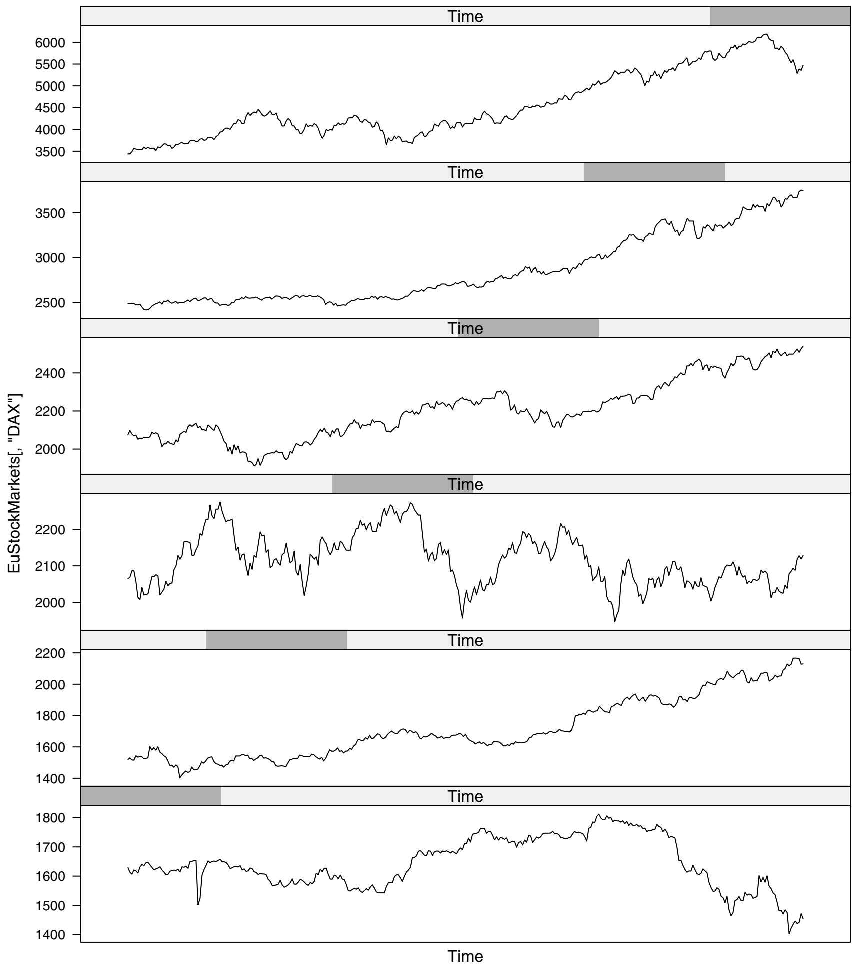

Figure 10.17. A cut-and-stack plot of stock market time-series data. The x variable is also used as a conditioning variable, with the effect of splicing up the x-axis across different panels. The y-axes are determined independently for each panel, emphasizing local variation rather than overall trends.

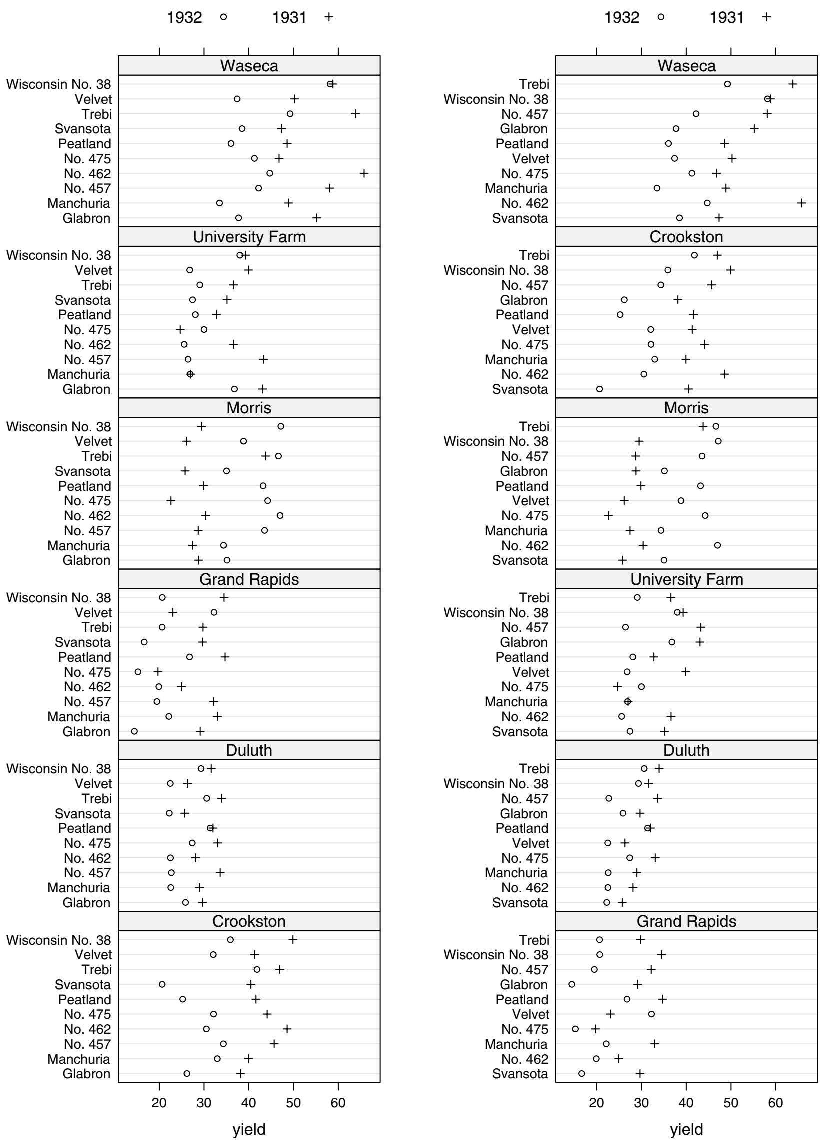

Figure 10.18. A grouped dot plot of the barley data with default (alphabetical by levels) and median (Figure 2.6) ordering. The ordered version makes it easier to convince ourselves that switching the year labels for Morris is enough to "fix" the data.

```
> dotplot(reorder(name, density) ~ 1000 * density, state.density,
          scales = list(x = list(log = 10)),
          xlab = "Density (per square mile)")
```

Taking logarithms alleviates the problem with Figure 10.15, and is generally preferable over artificial axis breaks. The inline call to `reorder()` creates a new factor with the same values and the same levels as `name`, but the levels are now ordered such that the first level is associated with the lowest value of `density`, the second with the next lowest, and so on. In this example there is exactly one value of `density` associated with each level of `name`, but this will not be true in general. In the following call, we reorder the levels of the `region` variable, a geographical classification of the states, again by `density`.

```
> state.density$region <-
      with(state.density, reorder(region, density, median))
```

Because there are multiple states for every region, we need to summarize the corresponding densities before using them to determine an order for the regions. The default summary is the mean, but here we use the median instead by specifying a third argument to `reorder()`.

Our eventual goal is to use `region` as a conditioning variable in a dot plot similar to the last one, but with states grouped by region. To do so, we first need to ensure that the levels of `name` for states within a region are contiguous, as otherwise they would not be contiguous in the dot plot.[6] This is achieved simply by reordering their levels by `region`. We would also like the states to be ordered by `density` within `region`, so we end up with another call to `reorder()` nested within the first one (this works because the original order is retained in case of ties).

```
> state.density$name <-
      with(state.density,
           reorder(reorder(name, density), as.numeric(region)))
```

We need to convert the `region` values to their numeric codes as the step of averaging would otherwise cause an error. Finally, we can use these reordered variables to produce the dot plot in Figure 10.20, with `relation = "free"` for the y-axis to allow independent scales for the different regions:

```
> dotplot(name ~ 1000 * density | region, state.density,
          strip = FALSE, strip.left = TRUE, layout = c(1, 4),
          scales = list(x = list(log = 10),
                        y = list(relation = "free")),
          xlab = "Density (per square mile)")
```

This still leaves room for improvement; the panels all have the same physical height, but different numbers of states, resulting in an odd looking plot. We could rectify this by changing the heights of the panels, as we did for widths in Figure 10.16. A convenience function that does this automatically, by making

[6] Factors are simply converted to the underlying numeric codes when they are plotted, and the codes are defined by the order of their levels.

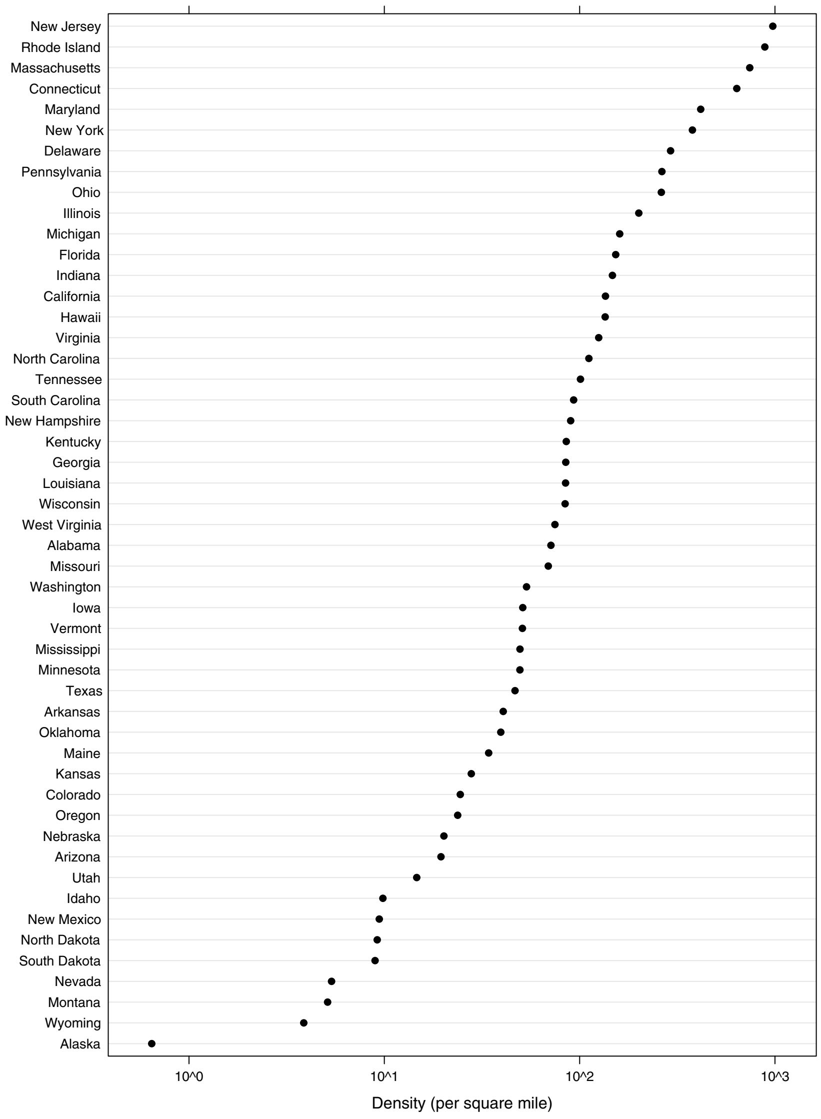

Figure 10.19. Population density in U.S. states, on a log scale. Comparing to Figure 10.15, we see that the states with the highest density no longer seem unusual compared to the rest. On the other hand, Alaska stands out as a state with unusually low density. Such judgments would have been harder if the states were not reordered.

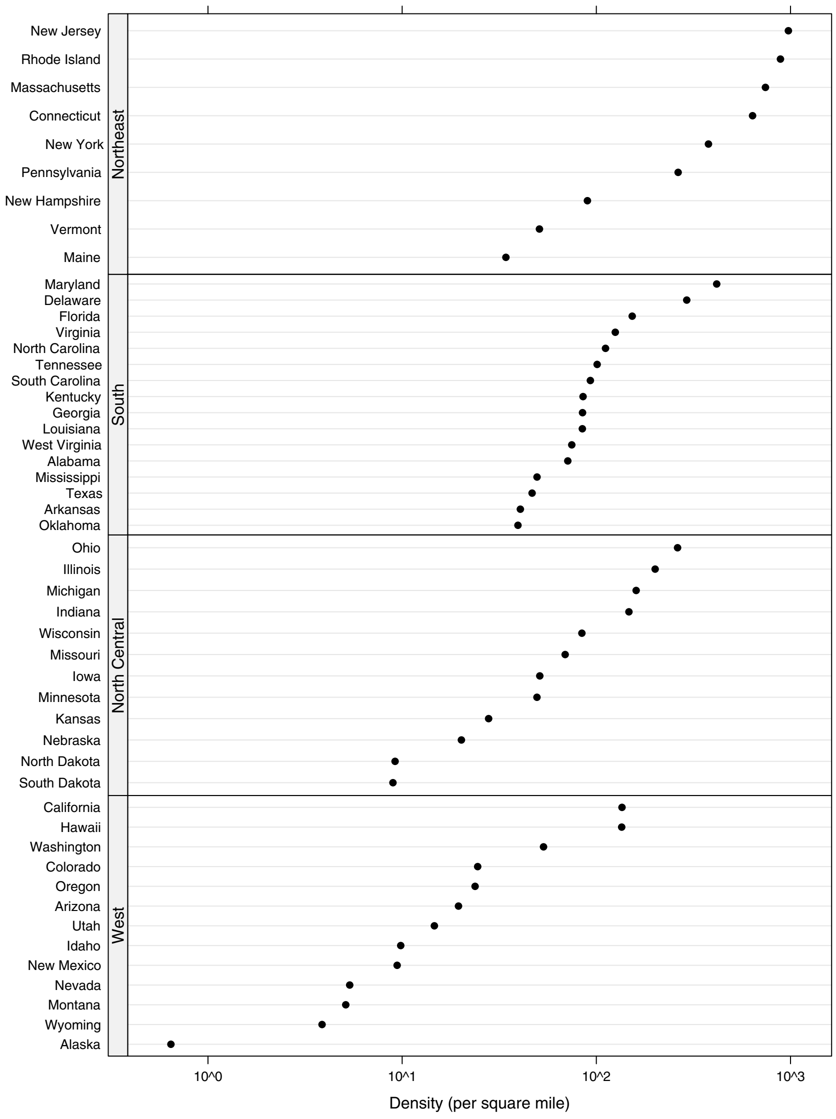

Figure 10.20. Population densities in U.S. states by region, with a different y-axis in each panel. Because the y variable is a factor, the scale calculations treat its values as the corresponding numeric codes; consequently, reordering of the levels is especially important. To appreciate this fact, the reader is strongly encouraged to try the call without reordering.

physical heights proportional to data ranges, is available in the latticeExtra package. Figure 10.21 is created by calling

```
> library("latticeExtra")
> resizePanels()
```

immediately after the previous `dotplot()` call. The inner workings of `resizePanels()` is explained in Chapter 12.

Another mode of automatic reordering is afforded by the `index.cond` argument, which we have used before without explanation to produce Figure 4.7. It implements a slightly different approach: it only reorders conditioning variables, and does so by trying to reorder packets based on their contents. To illustrate its use, consider the `USCancerRates` dataset in the latticeExtra package, which records average yearly deaths due to cancer in the United States between the years 1999 and 2003 at the county level. We plot the rates for men and women against each other conditioning by state, and order the panels by the median of the difference in the rates between men and women. Figure 10.22 is produced by

```
> data(USCancerRates, package = "latticeExtra")
> xyplot(rate.male ~ rate.female | state, USCancerRates,
         aspect = "iso", pch = ".", cex = 2,
         index.cond = function(x, y) { median(y - x, na.rm = TRUE) },
         scales = list(log = 2, at = c(75, 150, 300, 600)),
         panel = function(...) {
             panel.grid(h = -1, v = -1)
             panel.abline(0, 1)
             panel.xyplot(...)
         },
         xlab = "Deaths Due to Cancer Among Females (per 100,000)",
         ylab = "Deaths Due to Cancer Among Males (per 100,000)")
```

In general, `index.cond` can be a function, with a scalar numeric quantity as its return value, that is called with the same arguments as the panel function. When there is exactly one conditioning variable, its levels are reordered to put these return values in increasing order. For more than one conditioning variable, the order of each is determined by averaging over the rest.

10.7 Controlling the appearance of strips

Each panel in a multipanel lattice display represents a packet defined by a unique combination of levels of one or more conditioning variables. The purpose of the strips that typically appear above each panel is to indicate this combination. The contents of a strip can be customized using the `strip` argument. Strips can be placed to the left of each panel too, and these are controlled by the `strip.left` argument.

Both these arguments can be logical, with the corresponding strips suppressed if they are `FALSE` (`strip.left` is `FALSE` by default). The default

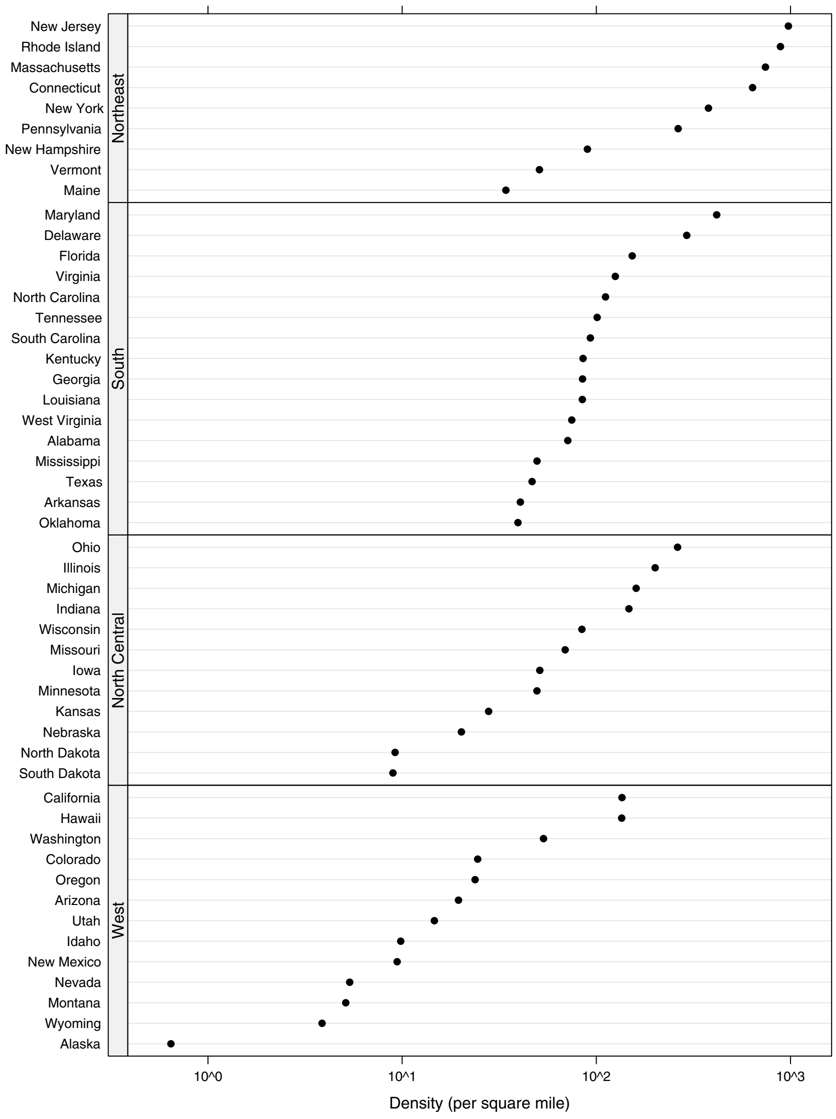

Figure 10.21. A variant of Figure 10.20, with panel heights varying by number of states.

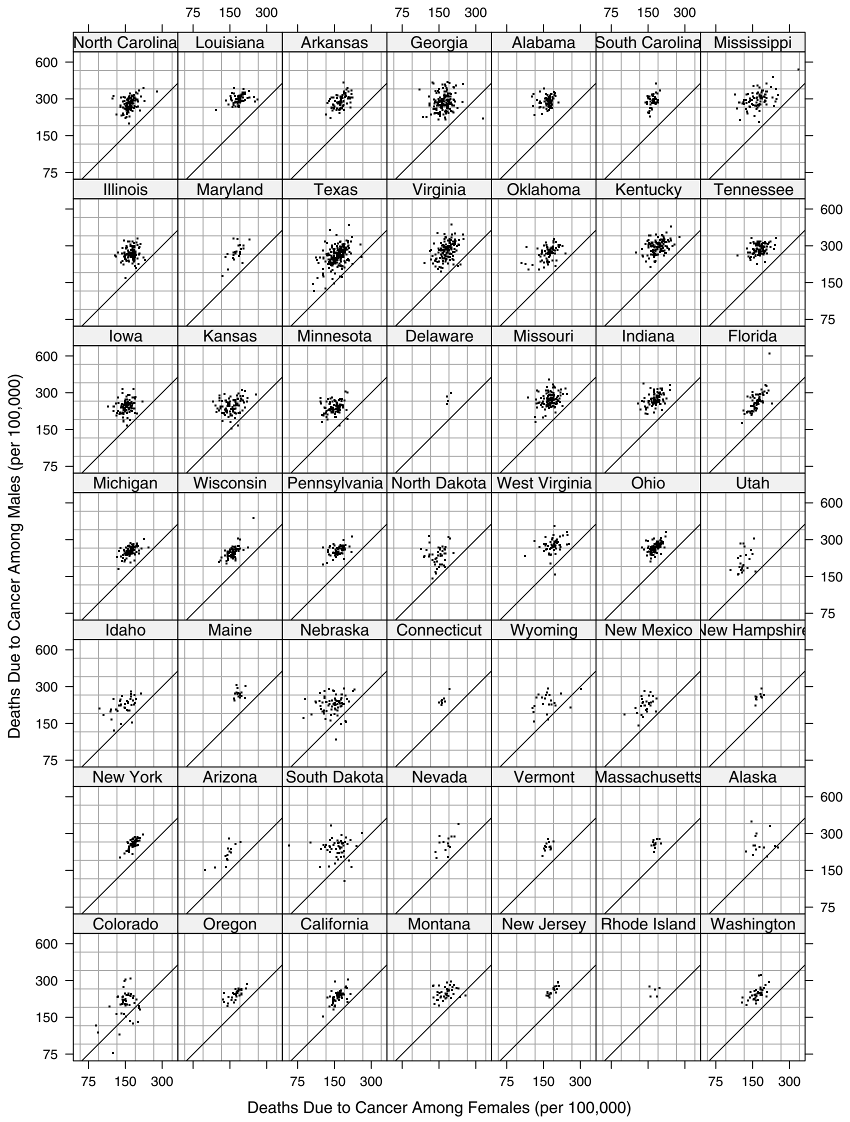

Figure 10.22. Annual death rates due to cancer (1999–2003) in U. S. counties by state for men and women, ordered by mean difference. A closer look reveals that the rate for women does not vary much across states, and the ordering is largely driven by the rate for men.

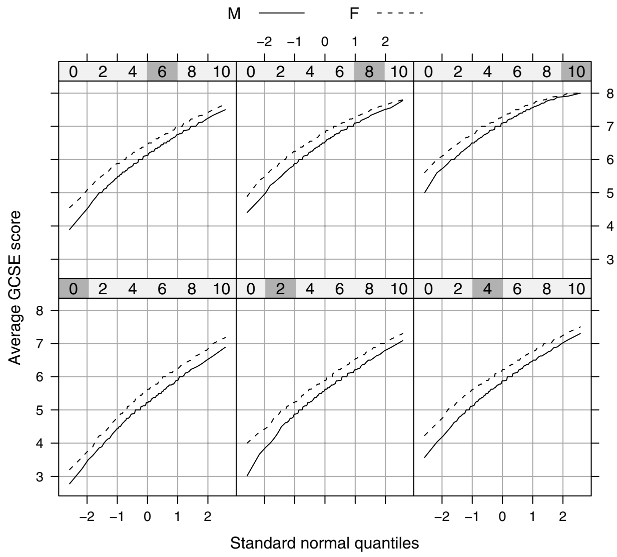

Figure 10.23. Normal Q–Q plots of average GCSE scores grouped by gender, conditioning on the A-level chemistry examination score, illustrating the use of a non-default strip annotation style. Another built-in style can be seen in Figure 11.5.

behavior can be changed by specifying the `strip` argument as a function that performs the rendering. One predefined function that can serve as a suitable strip function is `strip.default()`, which is used when `strip` is `TRUE`. `strip.default()` has several arguments that control its behavior, and these are often used to create variants. For example, one argument of `strip.default()` is `style`, which determines how the levels of a factor are displayed on the strip. We might define a new strip function as

```
> strip.style4 <- function(..., style) {
      strip.default(..., style = 4)
  }
```

When called, this function will call `strip.default()` with whatever arguments it received, with the exception of `style`, which will be changed to 4. This can be used to produce Figure 10.23 with

```
> data(Chem97, package = "mlmRev")
> qqmath(~gcsescore | factor(score), Chem97, groups = gender,
         type = c("l", "g"),  aspect = "xy",
         auto.key = list(points = FALSE, lines = TRUE, columns = 2),
         f.value = ppoints(100), strip = strip.style4,
         xlab = "Standard normal quantiles",
         ylab = "Average GCSE score")
```

Because it is common to create custom strip functions in this manner, a convenient generator function called `strip.custom()` is provided by lattice. `strip.custom()` is called with a subset of the arguments of `strip.default()` and produces another *function* that serves as a strip function by calling `strip.default()` after replacing the relevant arguments with their new values. Thus, an alternative call that produces Figure 10.23 is

```
> qqmath(~gcsescore | factor(score), Chem97, groups = gender,
         type = c("l", "g"),  aspect = "xy",
         auto.key = list(points = FALSE, lines = TRUE, columns = 2),
         f.value = ppoints(100), strip = strip.custom(style = 4),
         xlab = "Standard normal quantiles",
         ylab = "Average GCSE score")
```

where the user-defined strip function is concisely specified inline. A full list of the arguments that can be manipulated in this manner (and their effect) is given in the help page for `strip.default()`. The above description applies to `strip.left` as well. Strips on the left can be useful when vertical space is at a premium, as in Figure 10.17. It is rarely useful to have both sets of strips (however, see Figure 11.6 and the accompanying discussion).

Another argument that indirectly controls the contents of the strip is `par.strip.text`, which is supplied directly to a high-level call. It is usually a list containing graphical parameters (such as `col` and `cex`) that are meant to control the appearance of the strip text. In practice, the list is passed on to the strip function, which may or may not honor it (the default strip function does). In addition to graphical parameters, `par.strip.text` can also contain a parameter called `lines`, which specifies the height of each strip in multiples of the default. The following example illustrates its use in conjunction with a custom strip function that does not depend on `strip.default()`.

```
> strip.combined <-
      function(which.given, which.panel, factor.levels, ...) {
      if (which.given == 1) {
          panel.rect(0, 0, 1, 1, col = "grey90", border = 1)
          panel.text(x = 0, y = 0.5, pos = 4,
                     lab = factor.levels[which.panel[which.given]])
      }
      if (which.given == 2) {
          panel.text(x = 1, y = 0.5, pos = 2,
                     lab = factor.levels[which.panel[which.given]])
      }
  }
> qqmath(~ gcsescore | factor(score) + gender, Chem97,
         f.value = ppoints(100), type = c("l", "g"), aspect = "xy",
         strip = strip.combined, par.strip.text = list(lines = 0.5),
         xlab = "Standard normal quantiles",
         ylab = "Average GCSE score")
```

The `lines` component is used to halve the height of the strip, which would normally have occupied enough space for two strips. The actual strip function

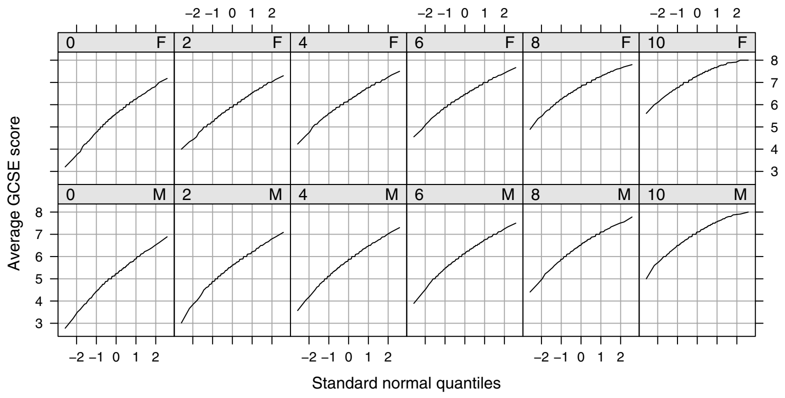

Figure 10.24. A variant of Figure 10.23, with gender now a conditioning variable as well, using a completely new strip function that incorporates both conditioning variables in a single strip.

is fairly transparent once we are told two facts: the strip functions for both conditioning variables use a common display area, and the native scale in the strip region is $[\,0, 1\,]$ in both axes. The resulting plot is shown in Figure 10.24.

10.8 An Example Revisited

Most of the examples we have seen in this book are designed to highlight some particular feature of lattice. Real life examples are typically more complex, requiring the use of many different features at once. Often, this simply results in a longer call. However, because of the way the various features of lattice interact, it is often possible to achieve fairly complex results with relatively innocuous looking code. To be able to write such code, one needs a familiarity with lattice that can only come from experience. We end this chapter with a study of one such example in some detail, with the hope that it will give the reader a sense of what can be achieved. The example is one we have encountered before; it was used to produce Figure 3.17:

```
> stripplot(sqrt(abs(residuals(lm(yield ~ variety+year+site)))) ~ site,
            data = barley, groups = year, jitter.data = TRUE,
            auto.key = list(points = TRUE, lines = TRUE, columns = 2),
            type = c("p", "a"), fun = median,
            ylab = expression(abs("Residual Barley Yield")^{1 / 2}))
```

The plot is based on the residuals from a linear model fit. Specifically, the square root of the absolute values of the residuals are plotted on the y-axis against one of the predictors (site) on the x-axis, with another predictor (year) used for grouping. The residuals are represented by points that are jittered horizontally to alleviate overlap, and lines joining their medians are added to summarize the overall trend. A legend describes the association between the graphical parameters used to distinguish levels of `year` and the actual levels.

To understand how we ended up with this call, let us consider how we might approach the task of producing the plot given this verbal description. The first step would be to fit an appropriate model, in this case

```
> fm <- lm(yield ~ variety + year + site, data = barley)
```

from which we could extract the residuals using

```
> residuals(fm)
```

Thus, the formula and data in the call might have been

```
> stripplot(sqrt(abs(residuals(fm))) ~ site, data = barley)
```

This is perfectly acceptable, but it runs the risk of a mismatch in the data used in the model fit and the plot. We instead choose to incorporate the model within the Trellis formula; the model is fit as part of the data evaluation step.

The next concern is the main display, which is controlled by the panel function. In this case, the display should consist of the (jittered) points for each group, along with a line joining the medians. Jittering is supported by the default panel function `panel.stripplot()`, through the `jitter.data` argument. However, a look at the help page for `panel.stripplot()` indicates no obvious way to add the lines, suggesting that we might need to write our own panel function. The predefined panel function `panel.average()` is ideal for our task; it even has an argument (`fun`) that can be used to specify the function that is used to compute the "average". Thus, our custom panel function might look like

```
panel = function(x, y, jitter.data, ...) {
    panel.stripplot(x, y, jitter.data = TRUE, ...)
    panel.average(x, y, fun = median, ...)
}
```

Now, according to Table 5.1, `panel.average()` can be invoked through `panel.xyplot()` by including `"a"` in the `type` argument. In addition, the help page for `panel.stripplot()` notes that it passes all extra arguments to `panel.xyplot()`. Add to this the fact that arguments unrecognized by `stripplot()` are passed along to the panel function, and we end up not requiring an explicit panel function at all, as long as we add the suitable arguments (`jitter.data`, `type`, and `fun`) to the high-level call. This also magically makes the adjustments required to accommodate the `groups` argument. Of course, such convenient panel functions are not always available, but they are often enough to make this approach useful.

The final piece that completes the plot is the legend. We make use of the `auto.key` argument, described in Chapter 9, which works by taking advantage of the fact that the default plotting symbols and line types are obtained from the global parameter settings. Since the output of `stripplot()` does not

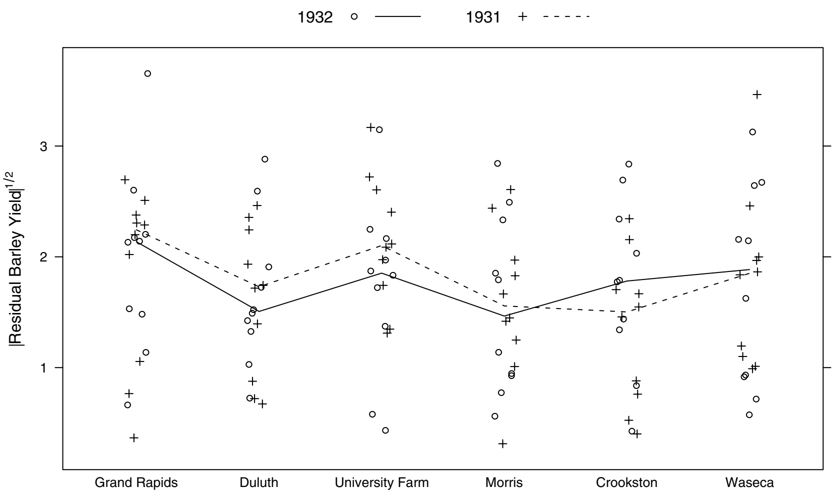

Figure 10.25. A spread–location plot of the barley data after "fixing" the Morris anomaly; compare with Figure 3.17.

normally contain lines, they are not included in the default legend, and we need to explicitly ask for them. Also notable is the use of `expression()` to specify the y-axis label. This is an example of the LaTeX-like mathematical annotation (Murrell and Ihaka, 2000) that can be used in place of plain text in most situations.[7] In Figure 10.25, we redraw Figure 3.17, after switching the values of `year` for the observations from Morris using

```
> morris <- barley$site == "Morris"
> barley$year[morris] <-
      ifelse(barley$year[morris] == "1931", "1932", "1931")
```

The call to produce the plot remains otherwise unchanged.

[7] See `?plotmath` for a general description of these capabilities.