8

Plot Coordinates and Axis Annotation

In this chapter, we discuss how the coordinate system for each panel is determined, how axes are annotated, and how one might control these in a lattice display. Control is possible at several levels, with a trade-off between the degree of control desired and the amount of effort required to achieve it.

8.1 Packets and the prepanel function

The controls discussed in this chapter can be broadly classified into two groups. Those in the first group relate to the determination of the coordinate system and axis limits for the panels (the rectangular regions within which graphics are drawn). Those in the second are concerned with how this coordinate system is described in the plot, typically through the use of tick marks and labels outside the panels. A grasp of the process determining the panel limits is essential to understand both sets of controls.

As described in Chapter 2, each combination of levels of the conditioning variables defining a *"trellis"* object gives rise to a *packet*. Loosely speaking, a packet is the data subset that goes into a panel representing a particular combination. Not all packets in a *"trellis"* object need end up in a plot of the object, and some may be repeated; however, a panel's limits are always determined by the entire collection of packets, and two panels with the same packet will have identical limits. In other words, limits are a property of packets, not panels. The rules determining these limits are described next.

Each panel area is a rectangular region in the Euclidean plane, and is defined completely by a horizontal and a vertical interval.[1] Even when the data being plotted are not intrinsically numeric (e.g., a categorical variable such as

[1] This is technically true even for functions such as `cloud()` and `splom()`, which are clearly different from other high-level functions, and bypass the controls described here. For these functions, scales and axis annotation are effectively controlled by the corresponding panel functions.

variety of oats), low-level plotting routines used to create the display require a numeric coordinate system. We refer to this as the *native coordinate system* of a panel. Given a packet, the prepanel function is responsible for determining a minimal rectangle in the native coordinate system that is sufficient to display that packet. It is implicitly assumed that any larger rectangle will also be sufficient for this purpose. Note that the minimal rectangle may depend not only on the packet but also on how it will eventually be displayed; for example, the maximum height of a histogram will differ greatly depending on whether it is a frequency, density, or relative frequency histogram, and to a lesser extent on the choice of bins.

Each high-level function comes with a default rule determining this minimal rectangle, and the `prepanel` argument gives the user further control. The rules governing the use of this argument are somewhat involved, and although the details are important, they are not immediately relevant. For the moment, assume that we have a rule to determine a minimal rectangle for each packet. A fuller discussion of the `prepanel` argument is postponed until later in this chapter.

8.2 The `scales` argument

8.2.1 Relation

There are three alternative schemes that prescribe, depending on how the panels are to relate to each other in the Trellis display, how the set of minimal rectangles collectively determines the final rectangles for each packet. The most common situation is to require all panels to have the same rectangle. This is achieved by choosing that common rectangle to be the one that minimally encloses all the individual rectangles. The second option is to allow completely independent rectangles, in which case the minimal rectangles are retained unchanged for each packet. The third option is to allow separate rectangles for each packet, but require their widths and heights (in the respective native coordinate systems) to be the same, with the intent of making differences comparable across panels, even if absolute positions are not (see Figure 10.6 for an example). In this case, each rectangle is expanded to make it as wide as the widest and as tall as the tallest rectangles. These rules can be selected by specifying `scales = "same"`, `scales = "free"`, and `scales = "sliced"`, respectively, in any high-level lattice call.

The description above is an oversimplification because in practice we often want to specify the relation between panels separately for the horizontal and vertical axes. This too can be achieved through the `scales` argument; for example,

```
scales = list(x = "same", y = "free")
```

leads to a common horizontal range and independent vertical ranges. More generally, the `scales` argument can also be used to specify a variety of other control parameters. In its general form, `scales` can be a list containing components called `x` and `y`, affecting the horizontal and vertical axes, each of which in turn can be lists containing parameters in `name = value` form. Parameters can also be specified directly as components of `scales`, in which case they affect both axes. For parameters specified both in `scales` and the `x` or `y` components, the values in the latter are given precedence.

As illustrated above, both `scales` and its `x` and `y` components can be a character string specifying the rule used to determine the packet rectangles. In the presence of other control parameters, this is no longer possible, and the string needs to be specified as the `relation` component. Thus, `scales = "free"` is equivalent to `scales = list(relation = "free")`, and

```
scales = list(x = "same", y = "sliced")
```

is equivalent to

```
scales = list(x = list(relation = "same"),
              y = list(relation = "sliced"))
```

Two other possible components of `scales` are involved in determining the panel coordinates, namely, `limits` and `axs`. It is difficult to discuss the purpose of these controls without first describing the prepanel function. For this reason, their discussion is likewise postponed until later in this chapter. Most other components of `scales` affect the drawing of tick marks and labels to annotate the axes. These are described next.

8.2.2 Axis annotation: Ticks and labels

Axis annotation is ultimately performed by the so-called *axis function*, which defaults to `axis.default()`, but can be overridden by the user. Other important functions under user control are ones that automatically determine tick mark locations and labels when these are not explicitly specified by the user. These functions, described later in this chapter, allow detailed control over axis annotation. However, such control is usually unnecessary, because some degree of control is already provided by components of the `scales` argument. We now list these components, noting that they apply only as long as the default axis annotation functions are used.

`log`

This parameter controls whether the data will be log-transformed. It can be a scalar logical, and defaults to `FALSE`, in which case no transformation is applied. If `log = TRUE`, the data are log-transformed with base 10. Other bases can be specified using a numeric value such as `log = 2`. The natural logarithm can be specified by `log = "e"`. The choice of base does not alter the panel display, but can affect the location and ease of interpretation of the tick marks and labels. The `log` component is ignored with a warning in certain situations (e.g., for factors).

A non-default value of `log` has two effects. First, the relevant primary variable is suitably transformed. This happens before it is passed to the prepanel and panel functions, which are in fact never aware of this transformation.[2] Second, this affects how the default axis labels are determined. Specifically, pretty tick mark locations are chosen in the transformed scale, but the labels nominally represent values in the original scale by taking the form `base^at`,[3] where `at` represents tick mark locations in the transformed scale.

`draw`

This parameter must be a scalar logical (`TRUE` or `FALSE`). If it is `FALSE`, the axes are not drawn at all, and the parameters described below have no effect.

`alternating`

This parameter is applicable only if `relation = "same"`. In that case, axes are not drawn separately for all panels, but only along the "boundary" of the layout. In other words, axes are drawn only along the bottom (respectively, top) of panels in the bottom- (top-) most row and the left (right) of panels in the left- (right-) most column.[4] Axis annotation can consist of tick marks and accompanying labels. The tick marks are always drawn (unless suppressed by other parameters), but labels can be omitted in a systematic manner using the `alternating` parameter. Specifically, `alternating` can be a numeric vector, each of whose elements can be `0`, `1`, `2`, or `3`. When it applies as a parameter in the `x` (respectively, `y`) component of `scales`, it is replicated to be as long as the number of columns (rows) in the layout. The values are interpreted as follows: for a row with value `0`, labels are not drawn on either side (left or right); for value `1`, labels are only drawn on the left; for value `2`, labels are only drawn on the right; and finally, for value `3`, labels are drawn on both sides. Similarly, for columns, values of `1` and `2` lead to labels on the bottom and top, `3` to labels on both sides, and `0` to labels on neither.

`alternating` can also be a logical scalar. `alternating = TRUE` is equivalent to `alternating = c(1, 2)` and `alternating = FALSE` to `alternating = 1`. This explains the name of the parameter; `alternating = TRUE` causes labels to alternate between bottom (left) and top (right) in successive columns (rows). This is the default for numeric axes, where it

[2] For example, `panel.lmline()` will fit a linear regression to the transformed values, which may not be what one expects.

[3] This is clearly not the best solution, but determining nice tick mark locations on a logarithmic scale is a difficult problem. See later sections for examples of alternatives.

[4] As a special case, axes are also drawn on the right of the last panel on the page, even if it is not in the rightmost column.

helps avoid overlapping labels in adjacent panels.

`tick.number`

This parameter acts as a suggested number of tick marks. Whether it will be used at all depends on the nature of the relevant variable; for example, it is honored for numeric (continuous) axes, but ignored for factors and shingles, because there is no reasonable basis for the selective omission of some labels in those cases.

`at`

The automatic choice of tick mark locations can be overridden using the `at` parameter. When `relation = "same"`, `at` should be a numeric vector specifying the tick mark locations, or `NULL`, which is equivalent to `at = numeric(0)`. When `relation = "free"` or `"sliced"`, `at` can still be a numeric vector (in which case it is used for all panels), but can also be a list. This list should be exactly as long as the number of packets. Each element can be a numeric vector or `NULL`. Alternatively, it could also be logical (both `TRUE` and `FALSE` are acceptable), in which case that particular packet falls back to the default choice of `at`.

The numeric locations of the tick marks must be specified in the native coordinates of the panel. This is true whether or not the axis is numeric. For factors and shingles, the ith level is usually encoded by the value i.

`labels`

By default, labels are chosen automatically to correspond to the `at` values. This default choice can be overridden using the `labels` parameter. Like `at`, it can be a vector, or a list when `relation` is not `"same"`. Labels can be character strings as well as "expressions", allowing LaTeX-like mathematical annotation (see `?plotmath`). If a component is logical, the default rule is used to determine labels for that packet. If the lengths of corresponding components of `at` and `labels` do not match, the result is undefined.

`abbreviate`

This is a logical flag, indicating whether the labels should be abbreviated using the `abbreviate()` function. Thic can be useful for long labels (e.g., for factors), especially on the x-axis.

`minlength`

This is passed on to the `abbreviate()` function if `abbreviate = TRUE`.

`format`

This is used as the `format` for *"POSIXct"* variables. See `?strptime` for a description of valid values.

`tck`

This parameter controls the length of tick marks, and is interpreted as a numeric multiplier for the default length. If `tck = 0`, ticks are not drawn at all. Negative values cause tick marks to face inwards, which is generally a bad idea, but is sometimes desired as a matter of style. `tck` can be a vector of length 2, in which case the first element affects the left (respectively, bottom) axis and the second affects the right (top) axis.

`rot`

This parameter can be used to specify an angle (in degrees) by which the axis labels are to be rotated. It can be a vector of length 2, to control left and right (bottom and top) axes separately.

`font`, `fontface`, `fontfamily`

These parameters specify the font for axis labels.

`cex`, `col`, `alpha`

These parameters control other characteristics of the axis labels. `cex` is a numeric multiplier to control character sizes. Like `rot`, it can be a vector of length 2, to control left and right (bottom and top) axes separately. `col` controls color and `alpha` controls partial transparency on some devices.

`col.line`, `alpha.line`, `lty`, `lwd`

These parameters control graphical characteristics of the tick marks. Note that `col.line` does not affect the color of panel boundaries, which may lead to unexpected results. However, parameters for tick marks and panel boundaries both default to the `"axis.line"` settings (see Chapter 7 for details), whereas parameters for labels default to the `"axis.text"` settings. Together, this gives explicit control over each component individually.

8.2.3 Defaults

The defaults for the components of `scales` may be different for different high-level functions. This is achieved through a special argument called `default.scales` which the casual user should not be concerned about except to realize the role it plays in determining the defaults. Any parameter specified in `default.scales` serves as the default value of the corresponding parameter in `scales`.[5] For the more common parameters, the global defaults (used when no value is specified in either `scales` or `default.scales`) are

[5] One important point to note is that parameters specific to a particular axis in `default.scales` can only be overridden by a similarly specific parameter in `scales`.

$$\begin{aligned}
\texttt{relation} &= \texttt{"same"} \\
\texttt{log} &= \texttt{FALSE} \\
\texttt{draw} &= \texttt{TRUE} \\
\texttt{alternating} &= \texttt{TRUE} \\
\texttt{tick.number} &= 5 \\
\texttt{abbreviate} &= \texttt{FALSE} \\
\texttt{minlength} &= 4 \\
\texttt{tck} &= 1 \\
\texttt{format} &= \texttt{NULL}
\end{aligned}$$

Most other parameters are graphical parameters that default to the settings active during plotting. One special case is `rot`, which defaults to `0` if `relation = "same"`, but for other values of `relation`, it defaults to `0` for the `x` component and `90` for the `y` component.

`default.scales` is used primarily in situations where one of the axes generally represents a categorical variable (factor or shingle). For such axes, the defaults change to

$$\begin{aligned}
\texttt{tck} &= 0 \\
\texttt{alternating} &= \texttt{FALSE} \\
\texttt{rot} &= 0
\end{aligned}$$

so that tick marks are omitted, the location of the labels do not alternate (saving space if the labels are long), and the labels are not rotated even when `relation` is not `"same"`. In `splom()`, `draw` defaults to `FALSE`, and much of the functionality of `scales` is accomplished instead by the `pscales` argument of `panel.pairs()`.

8.2.4 Three-dimensional displays: `cloud()` and `wireframe()`

The normal interpretation of “horizontal” and “vertical” axes makes no sense in the `cloud()` and `wireframe()` functions. There, the `scales` argument instead controls how the bounding box is annotated. As before, components can be specified directly, or in the `x`, `y`, or `z` components for specific axes. Many of the same parameters apply in this case, whereas many do not (and several should, but currently have no effect). There are two new parameters.

`arrows`

This parameter controls whether the annotation will be in the form of tick marks and labels, or just as an arrow encoding the direction of the axis. An arrow is used if `arrows = TRUE` (the default), and tick marks and labels are drawn otherwise. Both can be suppressed with `draw = FALSE`.

`distance`
: Labels describing the axes (`xlab`, `ylab`, and `zlab`) are drawn along edges of the bounding box. This parameter controls how far these labels are from the box. `distance` should be a scalar if it is specified in axis-specific components, but if specified as a component of `scales` directly, it should be (and is recycled if not) a vector of length 3, specifying distances for the `x`, `y`, and `z` labels.

8.3 Limits and aspect ratio

8.3.1 The prepanel function revisited

As briefly mentioned earlier, the prepanel function is responsible for determining a minimal rectangle big enough to contain the graphical encoding of a given packet. Because this graphic is produced by the panel function, the prepanel function has to be chosen in concordance with it, and may in principle require all the information available to the panel function. For this reason, the prepanel function is usually called with exactly the same arguments[6] as the panel function, once for every packet. An important distinction is that the prepanel function is called as part of the process creating the *"trellis"* object, whereas the panel function is only called during plotting.

The return value of the prepanel function determines the minimal bounding rectangle for a packet, but it can also affect the axis labels and aspect ratio. In full generality, the return value can be a list consisting of components `xlim`, `ylim`, `xat`, `yat`, `dx`, and `dy`. Each high-level function has a default rule to calculate these quantities, so a user-specified prepanel function is not required to return all of these components; any missing component will be replaced by the corresponding default. The interpretation and effect of these components are described below.

`xlim`, `ylim`
: These components together define a minimal bounding rectangle for the graphic to be created by the panel function given the same data packet. Two general forms are acceptable: a numeric vector of length 2 (as returned by `range()` for numeric data), and a character vector of arbitrary length (as returned by `levels()` for a factor). The first form is typical for numeric and date–time data, and `xlim` (or `ylim`, as the case may be) is interpreted as the horizontal (vertical) range of the rectangle. The second form is typical for factors, and is interpreted as a range containing `c(1, length(xlim))`, with the character vector determining labels at tick positions $1, 2, \ldots,$ `length(xlim)`. If no other explicit specification of limits is made (e.g., through the high-level arguments `xlim` and `ylim`, or the

[6] Actually, some arguments may be dropped if the function does not accept them, and the list may be different for prepanel and panel functions.

`limits` component of `scales`), then the actual limits of the panels are guaranteed to contain the limits returned by the prepanel function.

The prepanel function is responsible for providing a meaningful return value for these components when the data contain missing or infinite values, or when they are empty (zero length). When nothing else is appropriate, `xlim` and `ylim` should be `NA`.

The limits returned by the prepanel function are usually extended (or padded) by a certain amount (configurable through `lattice.options()`), to ensure that points on or near the boundary do not get clipped. This behavior may be suppressed by specifying `axs = "i"` as a component of `scales`. The default behavior corresponds to `axs = "r"`.

`xat`, `yat`
: When `xlim` (or `ylim`) is a character vector, this is taken to mean that the scale should include the first *n* integers, where *n* is the length of `xlim` (`ylim`). The elements of the character vector are used as the default labels for these `n` integers. Thus, to make this information consistent between panels, the `xlim` or `ylim` values should represent all the levels of the corresponding factor, even if some are not used within that particular panel. To make `relation = "free"` or `relation = "sliced"` behave sensibly in such cases, an additional component `xat` (`yat`) may be returned by the prepanel function, which should be a subset of `1:n`, indicating which of the `n` values (levels) are actually represented in the panel.

`dx`, `dy`
: The `dx` and `dy` components are numeric vectors of the same length, together defining slopes of line segments used for banking computations when `aspect = "xy"`, as described below.

8.3.2 Explicit specification of limits

The axis limits computed through the above mechanism can be overridden using the `xlim` and `ylim` arguments in high-level lattice functions. These should not be confused with the `xlim` and `ylim` components in the return value of `prepanel`, although they serve the same purpose and have the same valid forms. Specifically, the `xlim` and `ylim` arguments can either be numeric vectors of length 2, specifying an interval, or a character vector of length n, in which case the numeric data range is taken to be the interval $[1, n]$ with a suitable padding. As with the `at` and `labels` parameters of `scales`, `xlim` and `ylim` can also be specified on a per-packet basis when `relation = "free"`. In this case, they have to be lists, with each component a numeric or character vector as above, or `NULL` in which case the default limits are used for the corresponding packet. The value of `xlim` or `ylim` is ignored when the corresponding

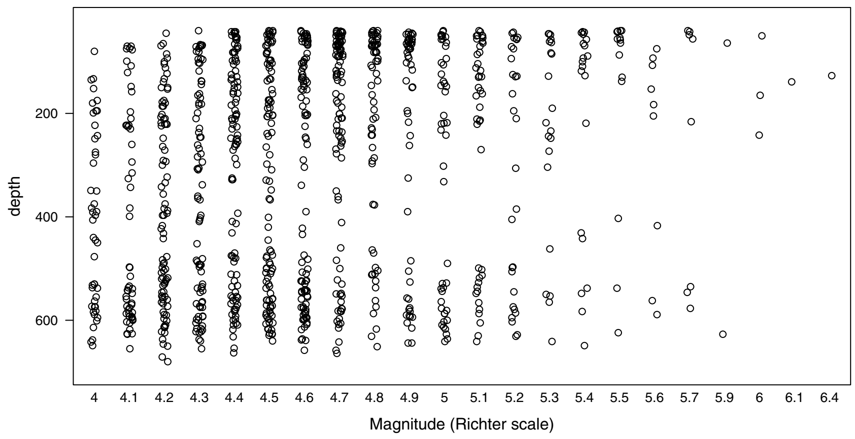

Figure 8.1. Rerendering of Figure 3.16, plotting the depth of earthquake epicenters against their magnitudes. The orientation of the $y$-axis has been reversed, so that depth now increases downwards.

`relation = "sliced"`. Alternatively, explicit limits can be specified as the `limits` components of `scales`. `scales$x$limits` is interpreted as `xlim` and `scales$y$limits` as `ylim`.

Although numeric limits usually take the form `c(minimum, maximum)`, this is not required. Limits in reverse order cause the corresponding axis to be reversed as well. For example, recall Figure 3.16, where `depth` was plotted on the vertical axis of a strip plot. Because depth is measured downwards from sea-level, it might be more natural to plot depth as increasing downwards in the graphic as well. This can be achieved with

```
> stripplot(depth ~ factor(mag), data = quakes, jitter.data = TRUE,
            ylim = c(690, 30),
            xlab = "Magnitude (Richter scale)")
```

An alternative that does not require knowing the data range beforehand is to make use of a custom prepanel function. Figure 8.1 is produced by

```
> stripplot(depth ~ factor(mag), data = quakes, jitter.data = TRUE,
            scales = list(y = "free", rot = 0),
            prepanel = function(x, y, ...) list(ylim = rev(range(y))),
            xlab = "Magnitude (Richter scale)")
```

Note that this will not work when `relation = "same"`, because in that case the process of combining limits from different packets makes the final limit sorted (even though there is only one packet in this example).

8.3.3 Choosing aspect ratio by banking

Statisticians are used to thinking in terms of invariance under scale and location changes, and often pay little attention to the unit of data being graphed. Although it usually makes no difference to the graphical encoding of the data, it is easy to see that the choice of units (including the choice of a base when taking logarithms) affects how the scales are annotated, indirectly affecting how easy it is to visually decode coordinates in the graphic. A less well-understood factor is the choice of physical units, that is, how much space (in centimeters, say) a plot will occupy on the display medium (computer monitor, paper, etc.). This is partly important because display media have finite physical resolution, and lines or points too close to each other will obscure patterns in the data. However, this problem is obvious when it occurs, and steps can be taken to rectify it. A more subtle feature is the ratio between physical units in the vertical (y) and horizontal (x) directions in a graphic, also known as the aspect ratio.

The importance of the aspect ratio has been noted by several authors (see Cleveland et al., 1988). In many situations, a satisfactory aspect ratio can only be found by trial and error. An exact aspect ratio, in the form of a numeric scalar, can be specified as the `aspect` argument. `aspect` can also be one of the character strings `"fill"`, `"iso"`, and `"xy"`. When `aspect = "fill"`, panels expand to fill up all available space.[7] When `aspect = "iso"`, the aspect ratio is chosen so that the relationship between the physical and data scales (units per cm) is the same on both axes. When `aspect = "xy"`, the aspect ratio is chosen using the 45° banking algorithm described by Cleveland et al. (1988), based on their observation that judgments about small changes in slope are made most accurately when the slopes are close to 45°. The exact calculations are performed by the `banking()` function,[8] assuming that the relevant slopes in the plot are defined by the `dx` and `dy` components returned by the prepanel function.

The default banking computation to choose the aspect ratio is particularly useful with time-series data, where order is important and the slopes of line segments joining successive points are meaningful. The following example uses the `biocAccess` dataset from the latticeExtra package, which records the number of hourly access requests to the `http://www.bioconductor.org` Web site during the months of January through May of 2007. Figure 8.2 is produced by

```
> data(biocAccess, package = "latticeExtra")
> xyplot(counts/1000 ~ time | equal.count(as.numeric(time),
                                          9, overlap = 0.1),
         biocAccess, type = "l", aspect = "xy", strip = FALSE,
```

[7] If necessary, initial layout calculations assume `aspect = 1` in this case.

[8] Actually, the function that performs the banking calculations is user-settable, and is obtained as `lattice.getOption("banking")`. The default is `banking()`, but this can be overridden to implement more sophisticated approaches to banking.

```
         ylab = "Numer of accesses (thousands)", xlab = "",
         scales = list(x = list(relation = "sliced", axs = "i"),
                       y = list(alternating = FALSE)))
```

Apart from banking, this example also illustrates the use of a date–time object as a primary variable, which affects how the x-axis is annotated.

For unordered data, the `dx` and `dy` components computed by the default prepanel function are less useful, and alternative computations may be more appropriate in some situations. For example, one might want to bank based on the slopes of a smoothed version of the data. This can be done using a custom prepanel function that computes a suitable smooth and returns `dx` and `dy` values computed from the smoothed curve. Such a prepanel function is available in lattice for LOESS smoothing, and is called `prepanel.loess()`. It can be used, with the `Earthquake` dataset, to produce Figure 8.3 as follows.

```
> data(Earthquake, package = "MEMSS")
> xyplot(accel ~ distance, data = Earthquake,
         prepanel = prepanel.loess, aspect = "xy",
         type = c("p", "g", "smooth"),
         scales = list(log = 2),
         xlab = "Distance From Epicenter (km)",
         ylab = "Maximum Horizontal Acceleration (g)")
```

Two other predefined prepanel functions are available in lattice. These are `prepanel.lmline()`, which is similar to `prepanel.loess()`, but fits a simple linear regression model instead, and `prepanel.qqmathline()`, used with `qqmath()`, which fits a line through the first and third quartile pairs.

8.4 Scale components and the axis function

Although the `scales` argument can contain parameters affecting both, axis annotation is in principle distinct from the determination of panel coordinates. Low-level control over annotation, beyond what is possible with `scales`, is provided through two mechanisms. The first is a pair of functions, supplied as the `xscale.components` and `yscale.components` arguments, that compute tick mark positions and labels. The second is the axis function, specified as the `axis` argument, that actually renders the annotation. The axis function typically uses the results of `xscale.components` and `yscale.components` as well as `scales`, but is not required to do so. Using custom replacements for `xscale.components` or `yscale.components` while retaining the default axis function has the advantage that the right amounts of space for the tick marks and labels are automatically allocated.

8.4.1 Components

One situation where the default choice of tick marks and labels can clearly be improved is when using logarithmic scales. We continue with the `Earthquake`

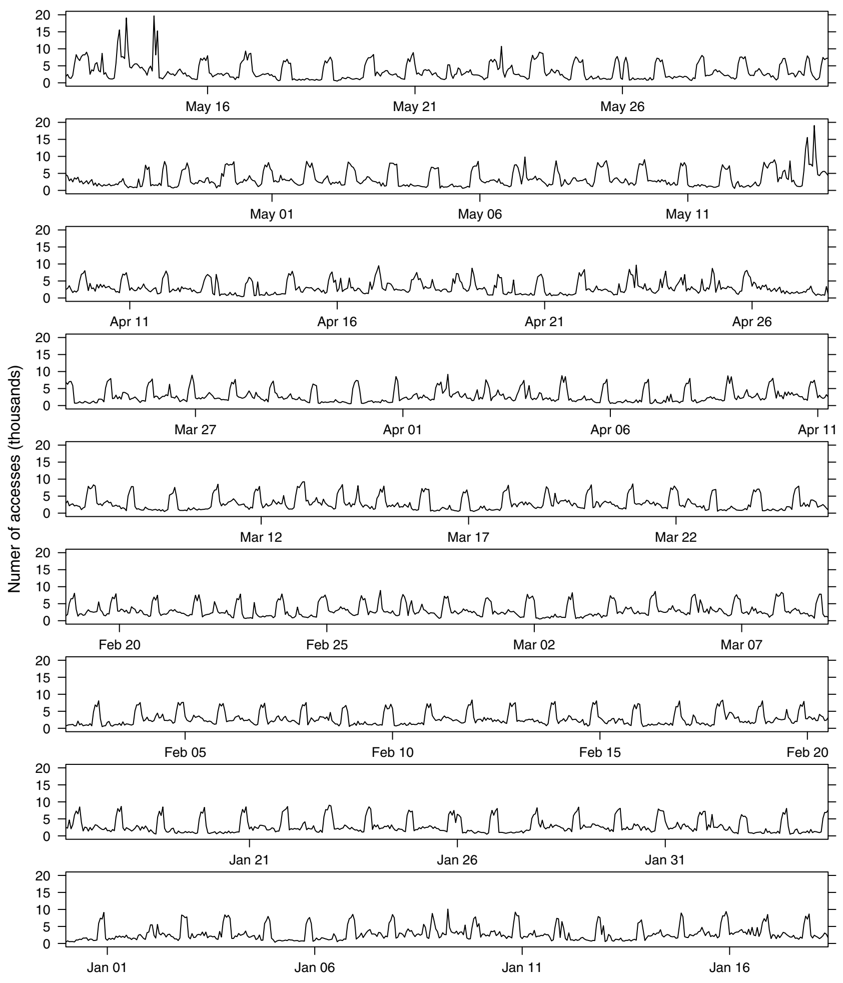

Figure 8.2. The hourly number of accesses to the `http://www.bioconductor.org` Web site during January through May of 2007. The aspect ratio has been chosen automatically using the 45° banking rule. The time axis has been split up into intervals to make use of the space available; such "cut-and-stack" plots are often useful with time-series data, and we encounter them again in Chapter 10. Figure 14.2 gives a more informative visualization of these data that makes effective use of some preliminary numerical analysis.

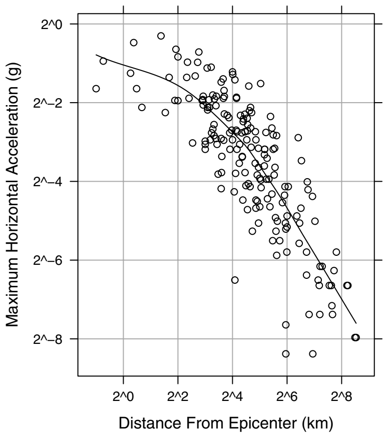

Figure 8.3. Rerendering of Figure 5.9, with automatically chosen aspect ratio. The predefined prepanel function `prepanel.loess()` is used, so that slopes of the LOESS smooth are used for the banking calculations. Note that the determination of the aspect ratio is unrelated to the display itself; if the `type` argument is omitted from the call, the display will not include the smooth, but the aspect ratio will remain unchanged.

example, with both x- and y-axes logarithmic, and try out some alternative ideas, implemented using the component functions. We only consider logarithms taken with base 2, but this can be adjusted as necessary.

Both `xscale.components` and `yscale.components` must return a list, with components `bottom` and `top` for the former, and components `left` and `right` for the latter, in addition to a component `num.limit` giving the numerical range of the limits as a vector of length 2. Details on the exact form of these components are not described here, but are available in the help page for `xscale.components.default()`. One interesting fact is that unlike `scales`, these allow the tick mark lengths to be vectorized, thus making major and minor tick marks possible.

In the following custom `yscale.components` function, we use the default components as a starting point, which allows us to bypass the uninteresting minutiae. We intend to have different labels on the two sides, so we next make the `right` component a copy of `left`. For both these components, the locations of the labels are kept unchanged, but the labels are modified. The labels on the left are turned into expressions, which leads to powers of 2 being rendered as superscripts. The labels on the right are converted into values in the original scale, possibly as fractions for negative powers, using the `fractions()` function from the MASS package. The final function is

```
> yscale.components.log2 <- function(...) {
      ans <- yscale.components.default(...)
      ans$right <- ans$left
      ans$left$labels$labels <-
          parse(text = ans$left$labels$labels)
      ans$right$labels$labels <-
          MASS::fractions(2^(ans$right$labels$at))
      ans
  }
```

A more ambitious approach is to determine tick mark locations afresh in the original scale, ignoring the default computations. We can adapt the `axTicks()` function for this purpose, to define a new function called `logTicks`, which takes a numeric range `lim`, and returns locations within the range that take the form $i \times 10^j$, where i takes the values specified in the `loc` argument.

```
> logTicks <- function (lim, loc = c(1, 5)) {
      ii <- floor(log10(range(lim))) + c(-1, 2)
      main <- 10^(ii[1]:ii[2])
      r <- as.numeric(outer(loc, main, "*"))
      r[lim[1] <= r & r <= lim[2]]
  }
```

This in turn can be used to define a custom `xscale.components` function:

```
> xscale.components.log2 <- function(lim, ...) {
      ans <- xscale.components.default(lim = lim, ...)
      tick.at <- logTicks(2^lim, loc = c(1, 3))
      ans$bottom$ticks$at <- log(tick.at, 2)
      ans$bottom$labels$at <- log(tick.at, 2)
      ans$bottom$labels$labels <- as.character(tick.at)
      ans
  }
```

Note that it suffices to change the `bottom` component, as the `top` component takes the same value by default. Both these custom replacements are used below to produce Figure 8.4.

```
> xyplot(accel ~ distance | cut(Richter, c(4.9, 5.5, 6.5, 7.8)),
         data = Earthquake, type = c("p", "g"),
         scales = list(log = 2, y = list(alternating = 3)),
         xlab = "Distance From Epicenter (km)",
         ylab = "Maximum Horizontal Acceleration (g)",
         xscale.components = xscale.components.log2,
         yscale.components = yscale.components.log2)
```

As noted earlier, the component functions allow tick mark lengths to be vectorized, making it fairly easy to add minor tick marks. Figure 8.5 gives an example of this feature, using a variant of the custom components function used earlier.

```
> xscale.components.log10 <- function(lim, ...) {
      ans <- xscale.components.default(lim = lim, ...)
      tick.at <- logTicks(10^lim, loc = 1:9)
```

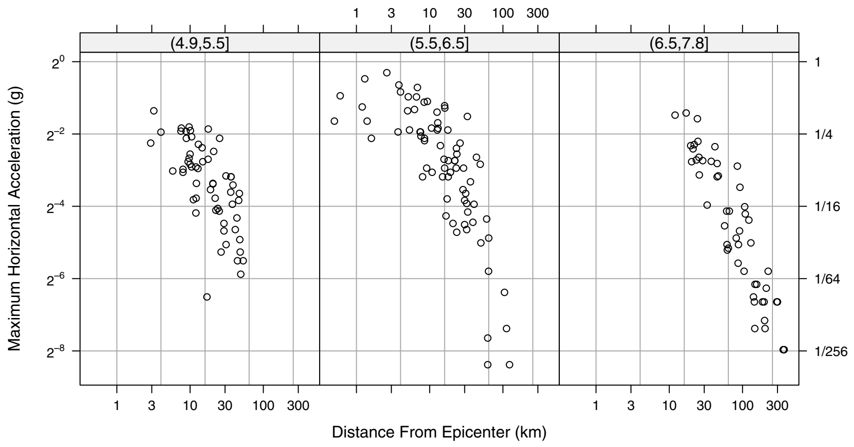

Figure 8.4. Fancy labels for logarithmic axes (compare with the axis annotation in Figure 8.3). The `alternating` parameter has been used to force axis labels on both the left and right sides simultaneously. The annotation is usually the same on both sides, but is different in this example where a user-supplied function has been used to compute the tick mark positions and labels.

```
        tick.at.major <- logTicks(10^lim, loc = 1)
        major <- tick.at %in% tick.at.major
        ans$bottom$ticks$at <- log(tick.at, 10)
        ans$bottom$ticks$tck <- ifelse(major, 1.5, 0.75)
        ans$bottom$labels$at <- log(tick.at, 10)
        ans$bottom$labels$labels <- as.character(tick.at)
        ans$bottom$labels$labels[!major] <- ""
        ans$bottom$labels$check.overlap <- FALSE
        ans
    }
> xyplot(accel ~ distance, data = Earthquake,
         prepanel = prepanel.loess, aspect = "xy",
         type = c("p", "g"), scales = list(log = 10),
         xlab = "Distance From Epicenter (km)",
         ylab = "Maximum Horizontal Acceleration (g)",
         xscale.components = xscale.components.log10)
```

Notice that logarithms are taken with base 10 in this example; the only effect this has on the panel display is to change the location of the reference grid lines.

8.4.2 Axis

Changing the components is not always enough, and sometimes one may want to take full control of axis drawing. One situation where this might be useful

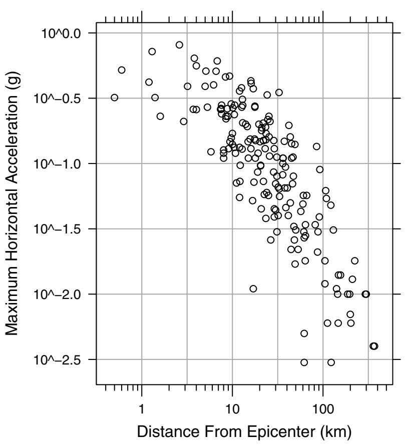

Figure 8.5. Another example of custom axis annotation: logarithmic axes with major and minor tick marks.

is when a single axis is used to represent multiple scales. Figure 8.6 plots a time-series of yearly temperatures in New Haven, CT, and annotates the temperature axis in both Celsius and Fahrenheit scales. This can be done using the following axis function, which uses `pretty()` to generate nice tick mark locations and `panel.axis()` (twice, in different colors) for the actual rendering. The same axis function must render the time axis as well, which our custom axis function handles simply by calling `axis.default()`.

```
> axis.CF <- function(side, ...) {
      if (side == "right") {
          F2C <- function(f) 5 * (f - 32) / 9
          C2F <- function(c) 32 + 9 * c / 5
          ylim <- current.panel.limits()$ylim
          prettyF <- pretty(ylim)
          prettyC <- pretty(F2C(ylim))
          panel.axis(side = side, outside = TRUE, at = prettyF,
                     tck = 5, line.col = "grey65", text.col = "grey35")
          panel.axis(side = side, outside = TRUE, at = C2F(prettyC),
                     labels = as.character(prettyC),
                     tck = 1, line.col = "black", text.col = "black")
      }
      else axis.default(side = side, ...)
  }
```

Figure 8.6 is produced by

```
> xyplot(nhtemp ~ time(nhtemp), aspect = "xy", type = "o",
         scales = list(y = list(alternating = 2, tck = c(1, 5))),
         axis = axis.CF, xlab = "Year", ylab = "Temperature",
```

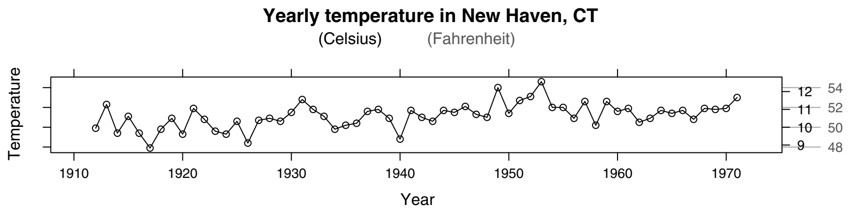

Figure 8.6. A custom axis function, providing calibration of temperature in both the Celsius and Fahrenheit scales. A legend has been added to describe the colors (see Chapter 9). Note the use of `tck` in `scales`. This affects the allocation of space for the tick marks and labels, which would otherwise need to be done manually.

```
main = "Yearly temperature in New Haven, CT",
key = list(text = list(c("(Celsius)", "(Fahrenheit)"),
           col = c("black", "grey35")), columns = 2))
```

One important point is that the axis function is called multiple times for each panel (once for each side), so careless use can easily lead to confusion. It should also be noted that the features discussed in the last section are fairly recent additions to lattice. Consequently, they are perhaps not as well thought out as the more traditional parts of the API, and some details may need to be changed in the future.