6

Trivariate Displays

Trivariate displays encode three primary variables in a panel. There are four high-level functions in lattice that produce trivariate displays: `cloud()` creates three-dimensional scatter plots of unstructured trivariate data, whereas `levelplot()`, `contourplot()`, and `wireframe()` render surfaces or two-dimensional tables evaluated on a systematic rectangular grid. Of these, `cloud()` and `wireframe()` are similar in that they both create two-dimensional projections of three-dimensional constructs, and they share several common arguments that control the details of the projection.

6.1 Three-dimensional scatter plots

We begin with `cloud()`, which produces three-dimensional scatter plots. Most of the discussion in this section about projection and how to control it in `cloud()` applies to `wireframe()` as well. We continue with the `quakes` example from the previous chapter. In Figure 5.6, we looked at a two-dimensional scatter plot of `lat` and `long`, with `depth` coded by grey level. The natural next step is to look at these in three dimensions. Figure 6.1 is produced by

```
> quakes$Magnitude <- equal.count(quakes$mag, 4)
> cloud(depth ~ lat * long | Magnitude, data = quakes,
        zlim = rev(range(quakes$depth)),
        screen = list(z = 105, x = -70), panel.aspect = 0.75,
        xlab = "Longitude", ylab = "Latitude", zlab = "Depth")
```

As before, we use the shingle `Magnitude` as a conditioning variable. The first part of the formula has a structure that is different from the ones we have encountered before. It has the form `z ~ x * y`, where `z` is the term plotted on the vertical axis, and `x` and `y` are plotted on the x- and y-axes. An equivalent form is `z ~ x + y`. This interpretation is also shared by the other high-level functions discussed in this chapter.

The `cloud()` function works by projecting points in three dimensions onto the two-dimensional display area. This is a fairly standard operation, and the

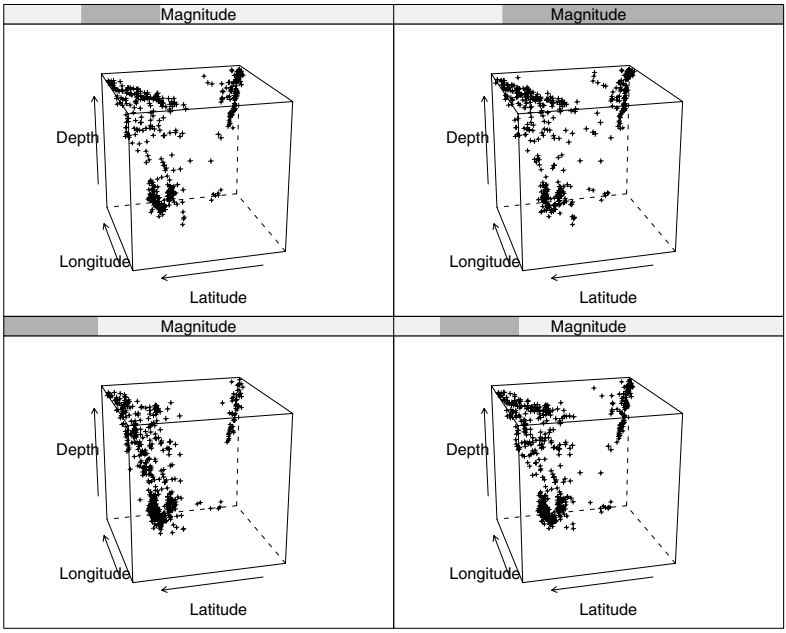

Figure 6.1. A three-dimensional scatter plot of earthquake epicenters in terms of latitude, longitude, and depth. Arrows indicate the direction in which the axes increase; the one for the `depth` is misleading because `zlim` has been reversed. A shingle derived from earthquake magnitude is used as a conditioning variable.

procedure is roughly as follows.[1] The first step is to determine a bounding box in three dimensions. By default, it is defined by the range of the data in each of the dimensions, but this can be changed by the `xlim`, `ylim`, and `zlim` arguments. The data are next centered and scaled, separately for each dimension. The center of the scaled bounding box is the origin, and the lengths of each side are usually the same. The latter can be controlled by the `aspect` argument, which in `cloud()` is a numeric vector of length 2. `aspect[1]` gives the ratio of the length of the scaled bounding box along the y-axis and that along the x-axis. Similarly, `aspect[2]` gives the ratio of lengths along the z- and x-axes. Note that this use of `aspect` is different from the normal use, which is to determine the aspect ratio of the panel. That purpose is served by the `panel.aspect` argument in this case.

The final step is to compute the two-dimensional projection. This is essentially defined by a viewpoint or "camera position" in three-dimensional space, in terms of the scaled coordinate system. Instead of being specified directly,

[1] These details are not strictly necessary for casual use, but are helpful in understanding some of the arguments we encounter later.

this viewpoint is determined by two arguments, `screen` and `distance`. `screen` defines the direction of the viewing point with respect to the origin, and `distance` the distance from it, determining the amount of perspective.

The direction is defined as a series of rotations of the bounding box. The viewpoint is initially set to a point on the positive z-axis, so that the positive x-axis points towards the right of the page, the positive y-axis points towards the top, and the positive z-axis is perpendicular to the page pointing towards the viewer. The bounding box, along with the data inside, can be rotated along any of these axes, one at a time, as many times as desired. The rotations are specified through the `screen` argument, which should be a list of named values, with names `x`, `y`, and `z` (each repeated 0 or more times), containing the amount of rotation in degrees. In the example above, we have `screen = list(z = 105, x = -70)`, which means that the bounding box was first rotated 105 degrees along the z-axis, followed by a rotation of -70 degrees along the x-axis. An alternative is to specify a 4×4 transformation matrix `R.mat` in homogeneous coordinates, to be applied to the data before `screen` rotates the view further. We do not go into the details of homogeneous coordinates as they are largely irrelevant; the important thing to know is that it is the de facto standard for specifying three-dimensional transformations and can thus be used to import a transformation from a different projection system. For example, the traditional graphics function `persp()` uses a different set of arguments to define a viewpoint, but its return value is a transformation matrix suitable for use as the `R.mat` argument. Conversely, the `ltransform3dMatrix()` function in lattice computes a suitable transformation matrix given a `screen` specification.

The other component defining the projection is perspective. Projections can be orthogonal, characterized by the feature that lines parallel in three-dimensional space remain parallel in the projection. Such plots can be obtained by setting the `perspective` argument to `FALSE`. Perspective projections are usually preferable, as they are a closer representation of how we view three-dimensional objects; specifically, distant objects are smaller and parallel lines appear to converge at a finite "horizon". The amount of perspective is determined by the `distance` argument, which is inversely related to the distance of the viewpoint from the center of the bounding box. Reasonable values of `distance` are between 0 and 1. Orthogonal projection can be thought of as viewing from an infinite distance,[2] and `distance = 0` is equivalent to `perspective = FALSE`.

We have already seen some of these arguments used in the previous example. We see a couple of new ones in the following call that produces Figure 6.2.

```
> cloud(depth ~ lat * long | Magnitude, data = quakes,
        zlim = rev(range(quakes$depth)), panel.aspect = 0.75,
        screen = list(z = 80, x = -70), zoom = 0.7,
```

[2] Through an infinitely powerful telescope that magnifies the view to fit our screen.

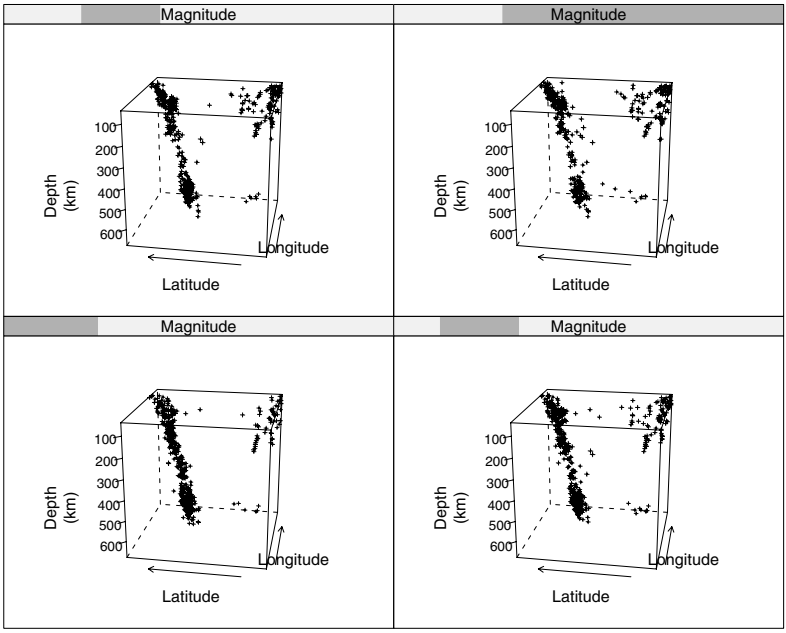

Figure 6.2. Another look at the locations of earthquake epicenters, from a different viewing direction and a few other variations. Together with Figure 6.1, this plot suggests that most of the epicenters are located along one of two distinct planes in three-dimensional space.

```
scales = list(z = list(arrows = FALSE, distance = 2)),
xlab = "Longitude", ylab = "Latitude",
zlab = list("Depth\n(km)", rot = 90))
```

In both examples, `zlim` is specified as an inverted range, so that depth values increase downward rather than upward. The scales are by default annotated by arrows indicating directions of the bounding box axes (which in the case of the z-axis in our example does not match the direction of the data axis). In the second example, we have used the `zoom` argument to shrink the plot slightly to make room for the axis labels, and the `scales` argument to replace the z-axis arrow by labeled tick marks. The default plotting character is a three-dimensional crosshair of sorts, consisting of three intersecting line segments, each parallel to one of the axes. The lengths of the segments are constant in three-dimensional space, but in a perspective projection the projected lengths depend on depth (those closer to the viewer are longer). In theory, this serves as a depth cue, although the benefits are negligible in practice. Other plotting characters can be specified using the `pch` argument, but the perspective transformation is not applied to them.

6.1.1 Dynamic manipulation versus stereo viewing

Projection-based three-dimensional displays benefit greatly from the ability to interactively manipulate details of the projection, such as the viewing direction. Not all features of the data are equally emphasized from all viewpoints, and it is extremely helpful to be able to choose one interactively. Unfortunately, lattice is implemented using a primarily static graphics paradigm, and support for interactive manipulation is sketchy at best. Even non-interactive manipulation, such as producing an animation by systematically moving the viewpoint in small increments, is helpful as the sense of motion it generates is a powerful cue for depth perception. This is possible with lattice in principle, but rendering is currently too slow for it to be practical. When such interaction is desired, alternative visualization systems such as GGobi (Swayne et al., 2003) and the OpenGL-based rgl package can prove to be much more effective unless lattice features such as conditioning are critical.

A couple of simple tricks can alleviate these problems to some extent. To get a comprehensive picture of the data, one can simultaneously view them from several angles. And although motion is not an option for static displays, stereo viewing can be almost as effective, although it does take some getting used to. The basic idea of stereo viewing is to simulate binocular vision by looking at two slightly different pictures through the two eyes; in particular, the one viewed by the right eye should be based on a viewpoint that is slightly to the right of the viewpoint defining the one seen by the left eye. In terms of the interface described above, this means that the "right eye" plot should be rotated clockwise along the y-axis by a small amount.

We combine both these ideas in Figure 6.3. Because the previous two plots suggest no strong dependence of the distribution of epicenters on earthquake magnitudes, we drop the conditioning variable. Our goal is thus to plot a packet containing the same data several times from different viewpoints. One way to implement this is to create separate *"trellis"* objects for each viewpoint and plot them one by one on the same page. A slightly less obvious approach, used here, is to take advantage of the indexing semantics of *"trellis"* objects. As we saw in Chapter 2, *"trellis"* objects can be indexed just as regular R arrays. In particular, an index can be repeated to repeat packets. We start by creating an object containing the data.

```
> p <-
      cloud(depth ~ long + lat, quakes, zlim = c(690, 30),
            pch = ".", cex = 1.5, zoom = 1,
            xlab = NULL, ylab = NULL, zlab = NULL,
            par.settings = list(axis.line = list(col = "transparent")),
            scales = list(draw = FALSE))
```

Next, we repeat it a suitable number of times and update it with a layout and a panel function that chooses a viewpoint depending on the position of the panel in the layout. Figure 6.3 is produced by

```
> npanel <- 4
> rotz <- seq(-30, 30, length = npanel)
> roty <- c(3, 0)
> update(p[rep(1, 2 * npanel)],
         layout = c(2, npanel),
         panel = function(..., screen) {
             crow <- current.row()
             ccol <- current.column()
             panel.cloud(..., screen = list(z = rotz[crow],
                                            x = -60,
                                            y = roty[ccol]))
         })
```

The current row and column are determined inside the panel function using the functions `current.row()` and `current.column()`, which we encounter more formally in Chapter 13. Rows in the figure represent different viewing directions, and columns differ by a small (three degrees) rotation along the y-axis. Viewing the result in stereo is somewhat nontrivial, but gets easier after the first time. The trick is to focus the eyes beyond the page, so that the figures on the left and the right column merge together. This process can be catalyzed by using a home-grown stereo viewer; roll up two pieces of paper tightly enough so that only one panel can be seen through each, then use one with each eye to look at different panels.

6.1.2 Variants and panel functions

Just as with other high-level functions, the default panel function in `cloud()` supports some variants of the standard display shown above, and the option of a user-supplied panel function provides further flexibility. In particular, the `groups` argument produces grouped displays as usual, and the `type` argument can be used to join the points by lines (`type = "l"`). Another useful value of `type` is `type = "h"`, which causes points to be joined to a "base" plane by vertical lines. Later in this chapter, we show how this feature could be used to create a three-dimensional bar chart of sorts. It can also be useful when absolute (rather than relative) values are being compared, as lengths are easier to compare than position after projection. In the following example, we plot the estimated population density in U.S. states (excluding Alaska and Hawaii) in 1975 as a function of their geographical "center". The data are available in separate R datasets, which we first need to collect together.

```
> state.info <-
      data.frame(name = state.name, area = state.x77[, "Area"],
                 long = state.center$x, lat = state.center$y,
                 population = 1000 * state.x77[, "Population"])
> state.info$density <- with(state.info, population / area)
```

Figure 6.4 is produced by

```
> cloud(density ~ long + lat, state.info,
        subset = !(name %in% c("Alaska", "Hawaii")),
```

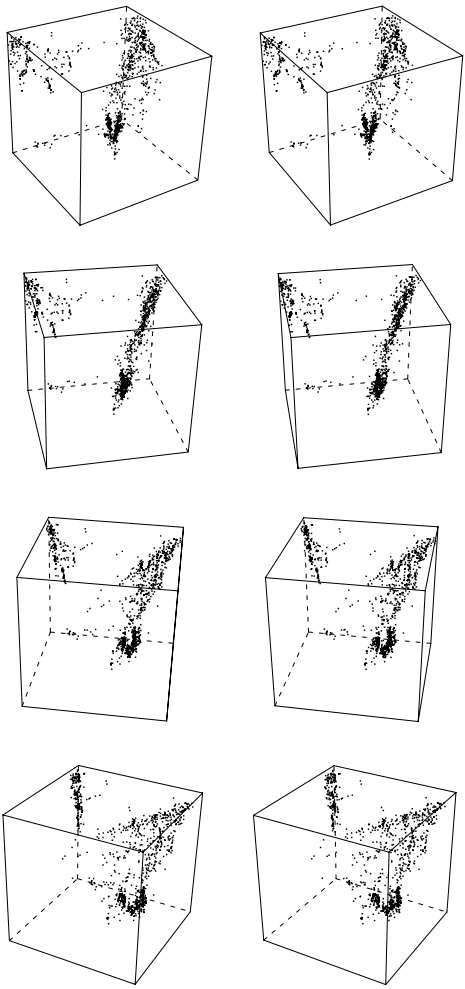

Figure 6.3. Unconditional three-dimensional scatter plots of earthquake epicenters, from different viewing directions. The rows represent different viewpoints, whereas columns differ only by a small rotation along the y-axis, simulating the difference between the positions of the left and right eyes. It is possible to achieve the illusion of depth by focusing the eyes on a point beyond the page and merging the two columns. The effect is often hard to achieve the first time, and it may help to look at the two columns separately through two pieces of rolled-up paper, creating a crude stereo viewer of sorts.

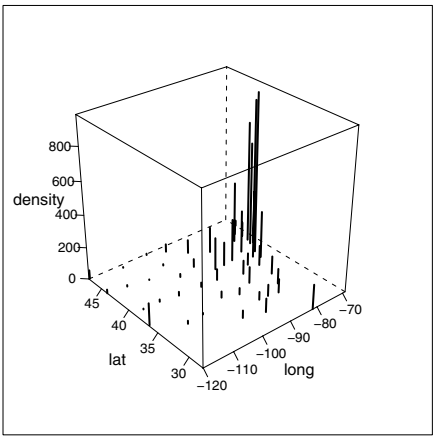

Figure 6.4. Population densities of U.S. states in 1975. Technically, the display is a three-dimensional scatter plot of densities on the z-axis plotted against approximate geographical centers of the states on the x–y plane. Line segments joining the points to their projections on the x–y plane encode the densities by length, making comparison easier. Although the overall spatial pattern is easily identifiable, it is difficult to associate the line segments with individual states. The aspect ratio could also be improved.

```
type = "h", lwd = 2, zlim = c(0, max(state.info$density)),
scales = list(arrows = FALSE))
```

A much more useful version of this plot is given in Figure 6.5, where state boundaries have been added to the bottom plane to serve as a reference. Creating such a plot is not difficult if we have access to state boundary data, but it requires some concepts we have not yet encountered; for this reason, the code to produce Figure 6.5 is postponed until Chapter 13.

6.2 Surfaces and two-way tables

The remaining trivariate functions in lattice are primarily intended for rendering surfaces and other array-like data, where the z-values are evaluated on a regular rectangular grid defined by the x- and y-values. In other words, the z-values form a matrix (at least conceptually), and the x- and y-values represent rows and columns of that matrix. Before going into the details of the individual functions, we discuss situations where they might be appropriate and how to prepare data for use with them.

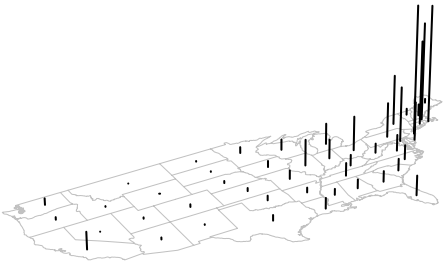

Figure 6.5. An improved version of Figure 6.4. A map of state boundaries on the x–y plane provides a useful visual reference. The aspect ratio is now more natural, and the distracting bounding box has been removed, along with the panel border.

6.2.1 Data preparation

Surfaces are different from other array-like data, as they are in principle smooth, or at least continuous, and can be abstractly represented as a function of two variables. However, they are conveniently represented as matrices containing evaluations of the function on a grid. Tables, on the other hand, are inherently discrete, and two-dimensional tables in particular are naturally represented as matrices. The visualization functions we discuss, namely `wireframe()`, `levelplot()`, and `contourplot()`, do not respect the distinction between surfaces and tables, and the user should be careful to use them in ways suitable for the data.

Before getting to the visualization step, one often has to preprocess the data to get them into a suitable form. We look at some typical examples before using them in plots. The most convenient situation is when the data are already evaluated on a grid, perhaps in the form of a matrix. Our first example, familiar to many R users, is the `volcano` data, which records the elevation of Maunga Whau (Mt. Eden), one of several extinct volcanos in the Auckland region, on a 10 m by 10 m grid. The data are in the form of a matrix, and there are no conditioning variables. Our next example is a correlation matrix, derived from data on car models on sale in the United States in 1993. The data are available in the MASS package.

```
> data(Cars93, package = "MASS")
> cor.Cars93 <-
      cor(Cars93[, !sapply(Cars93, is.factor)], use = "pair")
```

We exclude the categorical variables, although some of them could have been used for conditioning. Our third example is a multiway frequency table, using the Chem97 data again. We create a frequency table of score by gcsescore (discretized into ten equally sized groups) and gender.

```
> data(Chem97, package = "mlmRev")
> Chem97$gcd <-
      with(Chem97,
           cut(gcsescore,
               breaks = quantile(gcsescore, ppoints(11, a = 1))))
> ChemTab <- xtabs(~ score + gcd + gender, Chem97)
```

This of course creates a three-dimensional array, with the third dimension (gender) a natural conditioning variable. As with other lattice functions, it is helpful to convert this to a data frame, to be used with a formula. This can be done using the as.data.frame.table() function.[3]

```
> ChemTabDf <- as.data.frame.table(ChemTab)
```

Our last example is somewhat longer, and involves evaluating fitted regression surfaces on a regular grid. We use the environmental dataset (Bruntz et al., 1974; Cleveland, 1993), which consists of daily measurements of ozone concentration, wind speed, temperature, and solar radiation in New York City for 111 days in 1973. We fit regression models which predict ozone concentration, an indicator of smog, using the other measurements as predictors. As in the original analysis, we use cube root of ozone concentration as the response.

```
> env <- environmental
> env$ozone <- env$ozone^(1/3)
```

For purposes of conditioning, we could also create shingles from the predictors. For example, Radiation is used as a conditioning variable below to create the three-dimensional scatter plots in Figure 6.6.

```
> env$Radiation <- equal.count(env$radiation, 4)
> cloud(ozone ~ wind + temperature | Radiation, env)
```

A scatter-plot matrix is another useful visualization of the data; Figure 6.7 is produced by

```
> splom(env[1:4])
```

We next fit four regression models. The first model is a standard linear regression model. The remaining three are non-parametric; two are variants of LOESS (Cleveland and Devlin, 1988; Cleveland and Grosse, 1991), and the third uses local regression (Loader, 1999) from the locfit package.

```
> fm1.env <- lm(ozone ~ radiation * temperature * wind, env)
> fm2.env <-
      loess(ozone ~ wind * temperature * radiation, env,
```

[3] This also works for matrices such as volcano, although they can be used directly as well.

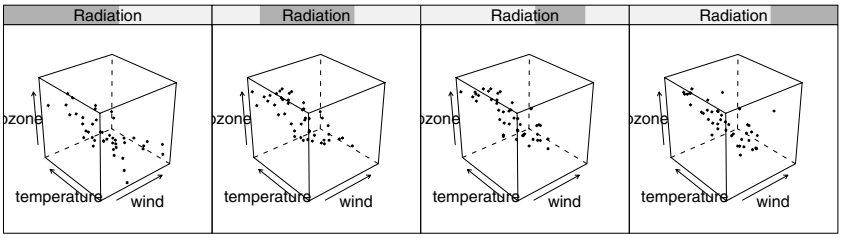

Figure 6.6. Conditional three-dimensional scatter plots showing the relationship among four continuous variables. The fourth variable, radiation, is used for conditioning.

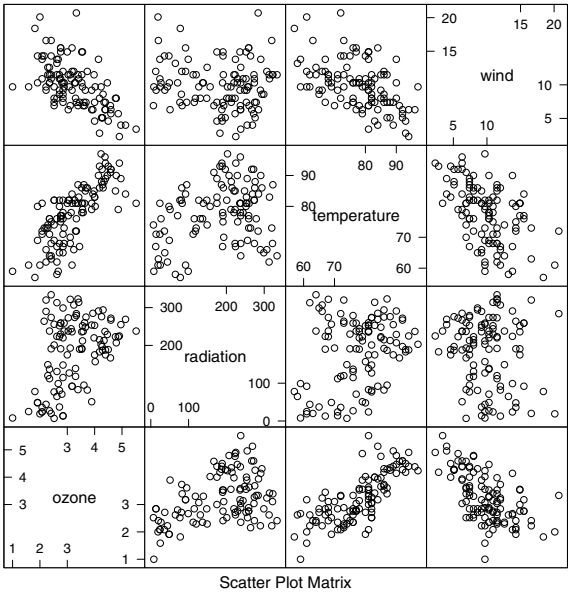

Figure 6.7. A scatter-plot matrix of ozone concentration, radiation, temperature, and wind speed. Neither this plot nor Figure 6.6 fully captures the four-dimensional relationship, but both are useful nonetheless. For our purposes, the most important feature is the correlation in certain pairwise scatter plots. For example, wind speed and temperature are negatively correlated, so there are no observations with high temperature and high wind speed.

```
            span = 0.75, degree = 1)
> fm3.env <-
      loess(ozone ~ wind * temperature * radiation, env,
            parametric = c("radiation", "wind"),
            span = 0.75, degree = 2)
> library("locfit")
> fm4.env <- locfit(ozone ~ wind * temperature * radiation, env)
```

Our eventual goal is to display the fitted regression surfaces. To do so, we first need to evaluate the predicted ozone concentrations on a regular grid of predictor values. There are three predictors, and we can only use two to define a surface, we therefore use one as a conditioning variable. We first create the vectors of values for each predictor that define the margins of the grid.

```
> w.mesh <- with(env, do.breaks(range(wind), 50))
> t.mesh <- with(env, do.breaks(range(temperature), 50))
> r.mesh <- with(env, do.breaks(range(radiation), 3))
```

The `expand.grid()` function can construct a full grid, in the form of a data frame, from these margins.

```
> grid <-
      expand.grid(wind = w.mesh,
                  temperature = t.mesh,
                  radiation = r.mesh)
```

The final step is to add columns in this data frame representing each of the fitted models. This can be easily done using the `predict()` methods for each of the fits.

```
> grid[["fit.linear"]] <- predict(fm1.env, newdata = grid)
> grid[["fit.loess.1"]] <- as.vector(predict(fm2.env, newdata = grid))
> grid[["fit.loess.2"]] <- as.vector(predict(fm3.env, newdata = grid))
> grid[["fit.locfit"]] <- predict(fm4.env, newdata = grid)
```

We now use these examples to create some plots.

6.2.2 Visualizing surfaces

We begin with the last example. Figure 6.8 is created with

```
> wireframe(fit.linear + fit.loess.1 + fit.loess.2 + fit.locfit ~
                                        wind * temperature | radiation,
            grid, outer = TRUE, shade = TRUE, zlab = "")
```

As with `cloud()`, the formula has the form `z ~ x * y`, but it is assumed in this case that `x` and `y` define a regular grid. Wind speed and temperature are used here as the x and y variables, and radiation as a conditioning variable. In this example, the formula actually contains four z variables separated by `+` signs. Normally, these would be used for grouping within each panel (as explained in Chapter 10), but the `outer = TRUE` argument causes them to be used for conditioning. By plotting all the fits together, we can compare

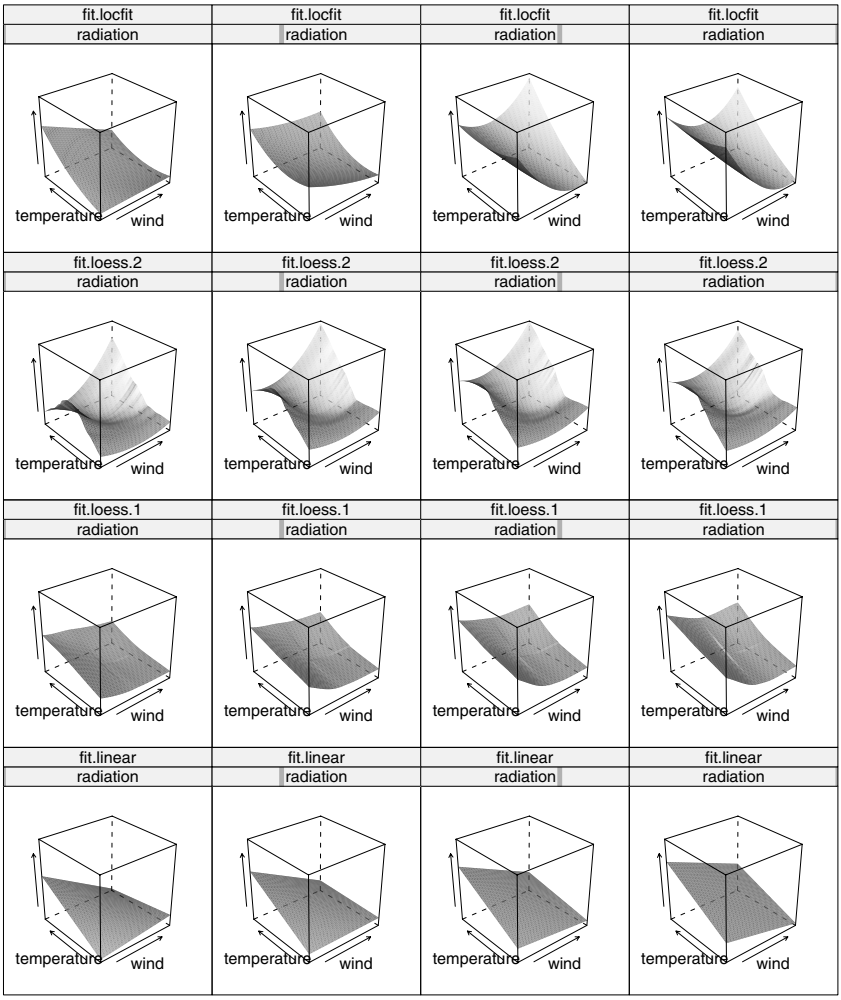

Figure 6.8. Wireframe plots of the fitted regression surfaces. Rows represent four different regression models and columns represent four levels of radiation; each panel graphs the surface representing predicted ozone concentration as a function of temperature and wind speed for a fixed level of radiation. The four models give widely inconsistent results when wind and temperature are both low or both high (the corners closest to and farthest from the viewer); these are regions where there are few actual observations.

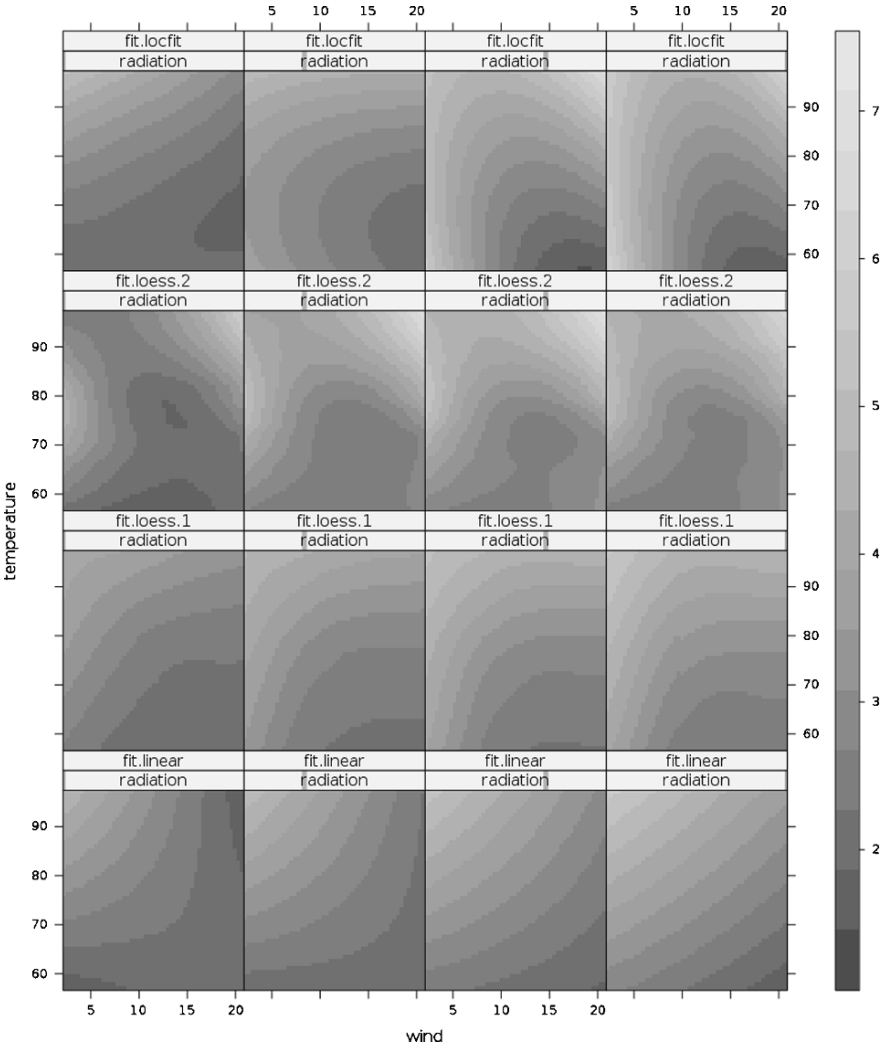

Figure 6.9. False-color level plots of fitted regression surfaces, in the same layout as Figure 6.8. This representation is independent of viewing direction, and easily conveys relative order between two points. Magnitudes of changes are harder to interpret without constantly referring to the color key. The choice of color can be important; in particular, greyscale gradients can only change in one direction, whereas true color gives more options. See the color plates for a color version of this figure.

their global characteristics. In each case, the predicted ozone levels generally increase with radiation. What is different is the behavior of the fitted surfaces as both wind speed and temperature increase. The reason for this can be understood if we look back at Figures 6.6 and 6.7; there are practically no observed data points with high values of both wind speed and temperature, and thus any regression fit will be unreliable in this region.

The `levelplot()` function has an interface identical to that of `wireframe()`, and works on the same type of data. However, instead of using projections, it uses a false-color gradient to encode the z variable. Figure 6.9 presents the same data as Figure 6.8 and is created by

```
> levelplot(fit.linear + fit.loess.1 + fit.loess.2 + fit.locfit ~
                                        wind * temperature | radiation,
            data = grid)
```

Yet another function with the same interface is `contourplot()`, which instead of using colors, draws contour lines. Generally, the contours are labeled by the level (value of the z variable) they represent, which may be preferable if the exact values are important; the disadvantage to this approach is that one cannot use too many levels, as then the labels tend to overlap. Figure 6.10 is created by

```
> contourplot(fit.locfit ~ wind * temperature | radiation,
              data = grid, aspect = 0.7, layout = c(1, 4),
              cuts = 15, label.style = "align")
```

All three functions have methods that work directly on a matrix. The following calls illustrate their use with the `volcano` data. The resulting plots are shown together in Figure 6.11.

```
> levelplot(volcano)
> contourplot(volcano, cuts = 20, label = FALSE)
> wireframe(volcano, panel.aspect = 0.7, zoom = 1, lwd = 0.5)
```

Note that the default of the `aspect` argument is different for these methods.

6.2.3 Visualizing discrete array data

Our other examples, a correlation matrix and a frequency table, represent data that are discrete in nature. `wireframe()` and `contourplot()`, which are designed for continuous surfaces, should not be used for such data. However, `levelplot()` can still be used, as we do in Figure 6.12, where grey levels are used to represent pairwise correlations between various continuous characteristics of several passenger car models for sale in the United States in 1993. Plots of correlation matrices are similar to scatter-plot matrices in structure, with individual scatter plots replaced by scalar summaries, namely the correlations. Figure 6.12 is produced by

```
> levelplot(cor.Cars93,
            scales = list(x = list(rot = 90)))
```

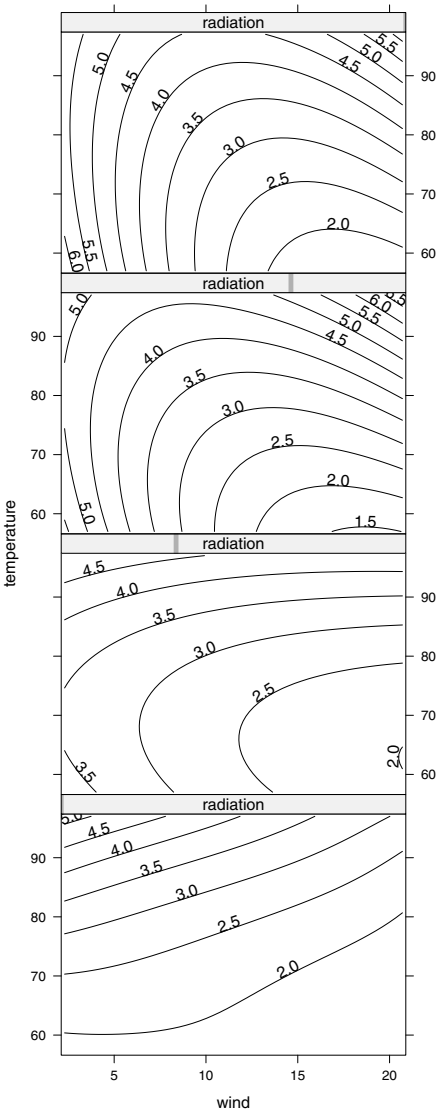

Figure 6.10. Contour plots of fitted regression surfaces. The design is similar to level plots, but shows the boundaries between levels rather than the levels themselves (although both can be combined in a single display). Contours can be labeled with the values they represent, enabling more direct decoding of the z variable. The density (closeness) of contour lines gives a sense of how fast the surface changes.

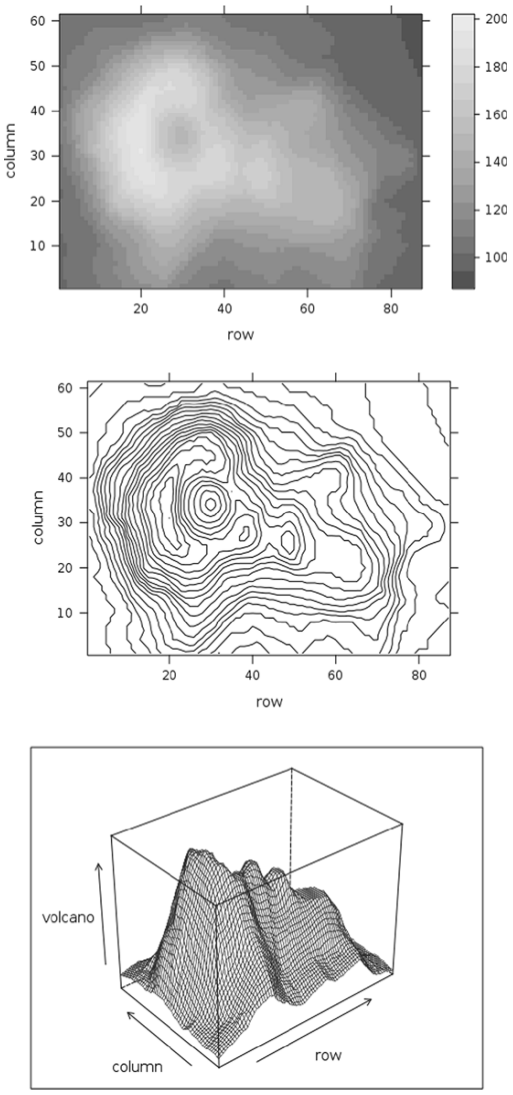

Figure 6.11. Visualizations of the topography of Mt. Eden, an extinct volcano in the Auckland region. The elevation values are stored as a matrix in the `volcano` dataset, which is used in the *"matrix"* methods for `levelplot()`, `contourplot()`, and `wireframe()` to produce the displays here.

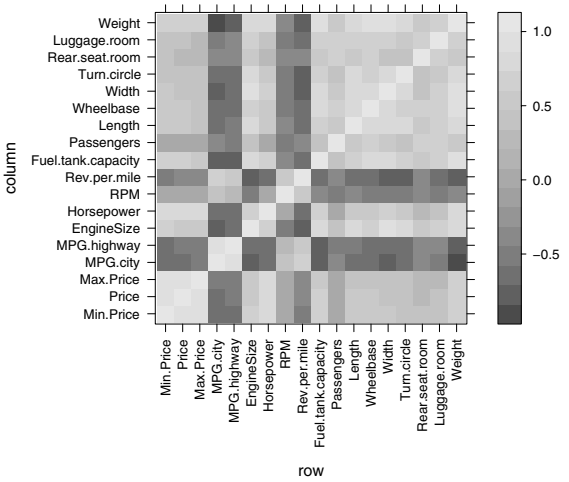

Figure 6.12. A correlation matrix derived from the `Cars93` data, visualized as a false-color image with grey levels encoding correlation. It would have been useful to be able to detect zero correlation easily, but this is not possible using grey levels alone. In addition, the order of rows and columns is arbitrary, making it difficult to see any patterns.

In this example, the order of rows and columns is arbitrary, and as with other types of plots, reordering them in a systematic manner can be helpful. One simple way to reorder rows or columns of a matrix is to first cluster them, and then order them in a manner consistent with the clustering. In the following example, we do so using the `hclust()` function. We also specify an explicit vector of levels where the colors change, instead of using the range of the correlations.

```
> ord <- order.dendrogram(as.dendrogram(hclust(dist(cor.Cars93))))
> levelplot(cor.Cars93[ord, ord], at = do.breaks(c(-1.01, 1.01), 20),
            scales = list(x = list(rot = 90)))
```

The resulting plot is shown in Figure 6.13. Other displays of correlation matrices, such as those described by Friendly (2002), can be produced from similar data using custom panel functions; two examples can be seen in Figures 13.5 and 13.6.

The frequency table derived from the `Chem97` data can be similarly visualized using `levelplot()`. It is often helpful to encode frequencies after taking their square root. To do so with a color gradient, we must also modify the color key to reflect this transformation. Figure 6.14 is produced by

```
> tick.at <- pretty(range(sqrt(ChemTabDf$Freq)))
> levelplot(sqrt(Freq) ~ score * gcd | gender, ChemTabDf,
```

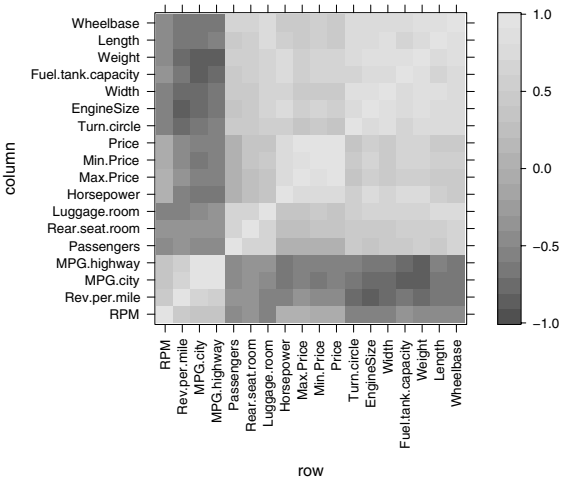

Figure 6.13. A slightly modified version of Figure 6.12, with rows and columns reordered according to a hierarchical clustering. Similar columns are now easily identifiable. This does not address the problem of emphasizing strength and direction of correlation separately, which is considered later in Chapter 13.

```
    shrink = c(0.7, 1), aspect = "iso", colorkey =
        list(labels = list(at = tick.at, labels = tick.at^2)))
```

In addition to a color gradient, this example also encodes the z-values using the size of the rectangles. The details are controlled by `shrink`, which is an argument of the default panel function `panel.levelplot()`.

Although `wireframe()` is unsuitable for discrete data, they can still be plotted in a three-dimensional projection using `cloud()`, ignoring the regular structure in the x- and y-values. For example, the following code would create a three-dimensional bar chart of sorts where frequencies are encoded by line segments.

```
> cloud(Freq ~ score * gcd | gender, data = ChemTabDf, type = "h",
        aspect = c(1.5, 0.75), panel.aspect = 0.75)
```

We do not show the results of this call, but instead use the `panel.3dbars()` function available in the latticeExtra package to create more "solid" versions of the bars. Figure 6.15 is produced by

```
> library("latticeExtra")
> cloud(Freq ~ score * gcd | gender, data = ChemTabDf,
        screen = list(z = -40, x = -25), zoom = 1.1,
        col.facet = "grey", xbase = 0.6, ybase = 0.6,
        par.settings = list(box.3d = list(col = "transparent")),
        aspect = c(1.5, 0.75), panel.aspect = 0.75,
        panel.3d.cloud = panel.3dbars)
```

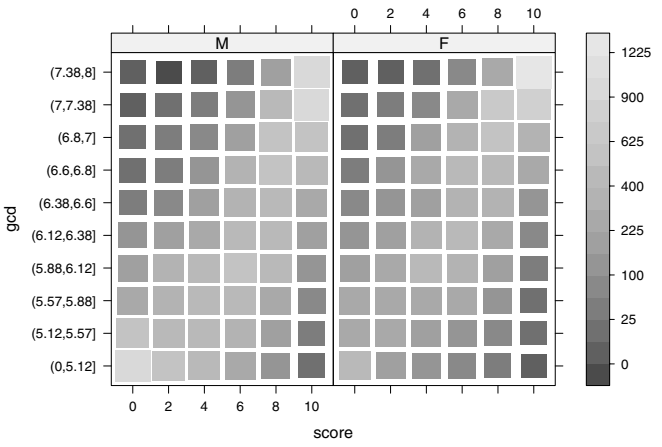

Figure 6.14. False color plots of a table derived from the `Chem97` data, showing the relationship between `score` and `gcsescore`, conditioning on `gender`. Counts are encoded by grey level as well as size of the rectangles. As with Figure 6.9, using color instead of grey levels considerably increases the usefulness of the display; in particular, the rectangles on the lower-left and upper-right corners are almost the same color as the background, making them difficult to see.

Note that we have specified a `panel.3d.cloud` argument rather than a `panel` argument; this is because the panel function in `cloud()` and `wireframe()` are responsible for computing the projections and drawing the bounding box, a task we normally wish to leave unchanged. The data-dependent part of the display is the responsibility of the `panel.3d.cloud` argument (`panel.3d.wireframe` for `wireframe()`) of `panel.cloud()`, and this is the piece we replace in our example.

6.3 Theoretical surfaces

The methods we used to plot regression surfaces using `wireframe()` can be easily adapted to mathematical surfaces. For our next example, we consider four bivariate copulas (Nelsen, 1999), which are essentially joint distributions on the unit square with uniform marginals. Our goal is to plot the corresponding density functions, as computed by the `dcopula()` function in the `copula` package (Yan and Kojadinovic, 2007). We start, as before, by defining a grid and adding columns to it:

```
> library("copula")
> grid <-
      expand.grid(u = do.breaks(c(0.01, 0.99), 15),
                  v = do.breaks(c(0.01, 0.99), 15))
```

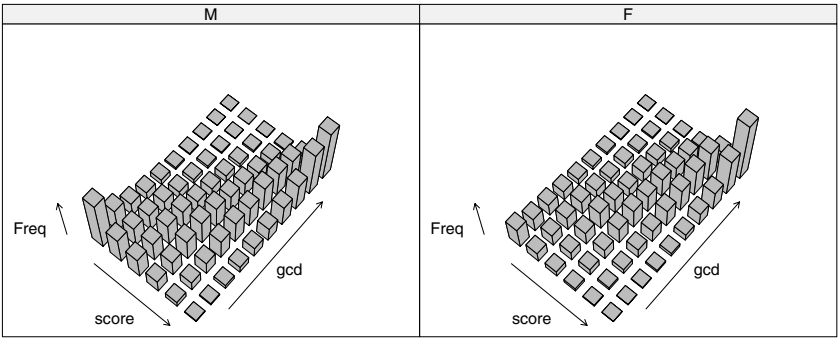

Figure 6.15. A three-dimensional bar chart showing the same data as Figure 6.14. The usefulness of such plots, compared to level plots, is questionable, as the information perceived depends to a considerable extent on choices that are not data-related, such as the viewing direction.

```
> grid$frank  <- with(grid, dcopula(frankCopula(2),    cbind(u, v)))
> grid$gumbel <- with(grid, dcopula(gumbelCopula(1.2), cbind(u, v)))
> grid$normal <- with(grid, dcopula(normalCopula(.4),  cbind(u, v)))
> grid$t      <- with(grid, dcopula(tCopula(0.4),      cbind(u, v)))
```

Figure 6.16 is now produced by

```
> wireframe(frank + gumbel + normal + t ~ u * v, grid, outer = TRUE,
            zlab = "", screen = list(z = -30, x = -50), lwd = 0.5)
```

The densities appear almost flat, as the vertical axis is dominated by changes close to the corners, even though we left out the corners themselves. In Figure 6.17, we try plotting the log-transformed densities, with better results.

```
> wireframe(frank + gumbel + normal + t ~ u * v, grid, outer = TRUE,
            zlab = "", screen = list(z = -30, x = -50),
            scales = list(z = list(log = TRUE)), lwd = 0.5)
```

Instead of transforming each term in the formula separately, we use the `scales` argument, which is described in detail in Chapter 8. Needless to say, these surfaces can also be visualized using `levelplot()`.

6.3.1 Parameterized surfaces

The surfaces we have seen thus far are defined as $z = f(x, y)$, where x and y vary over a continuous interval, approximated by a discrete grid. A more

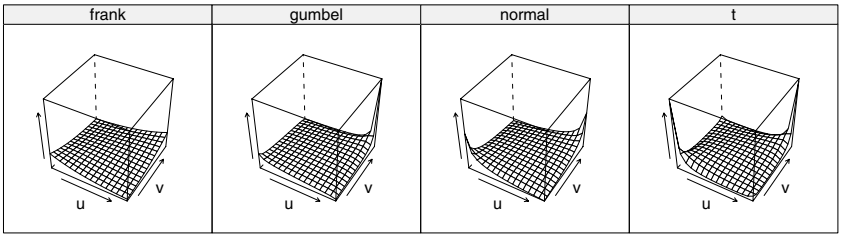

Figure 6.16. A wireframe plot representing the probability density function of four copulas. The surfaces appear to be largely flat because some of the densities increase rapidly close to the corners.

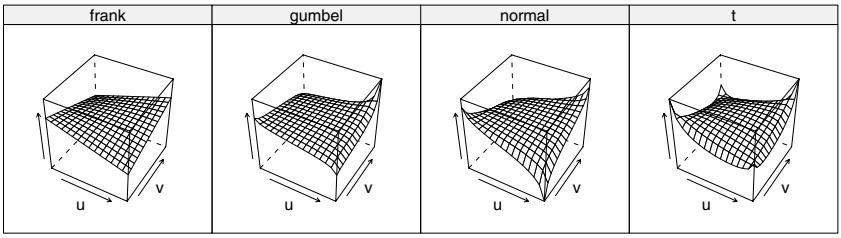

Figure 6.17. Figure 6.16 rerendered with log densities on the z-axis. The surfaces are now much easier to compare.

general way of representing surfaces is to parameterize them as functions of the form

$$\boldsymbol{f} : [\,0,1] \times [\,0,1] \longrightarrow \mathbb{R}^3$$

where every point on the surface corresponds to a point (u, v) on the unit square, with coordinates in three-dimensional space given by

$$\boldsymbol{f}(u,v) = (x(u,v), y(u,v), z(u,v))$$

A simple example of a parameterized surface is a sphere, which can be represented by the equations

$$\begin{aligned} x(\theta,\phi) &= \cos\theta\cos\phi \\ y(\theta,\phi) &= \sin\theta\cos\phi \\ z(\theta,\phi) &= \sin\phi \end{aligned}$$

where $\theta, \phi \in [-\pi, \pi]$ can be thought of as longitude and latitude, respectively (we use this interpretation later to create Figure 13.9). The domain here is not the unit square, but this can be easily rectified by a simple scale and location

shift. A somewhat more complicated, but fairly well-known example is the "figure 8" immersion of the Klein bottle, with a possible parameterization given by

$$
\begin{aligned}
x &= \cos u \ \left(r + \cos\frac{u}{2} \cdot \sin tv - \sin\frac{u}{2} \cdot \sin 2tv\right) \\
y &= \sin u \ \left(r + \cos\frac{u}{2} \cdot \sin tv - \sin\frac{u}{2} \cdot \sin 2tv\right) \\
z &= \sin\frac{u}{2} \cdot \sin tv + \cos\frac{u}{2} \cdot \sin tv
\end{aligned}
$$

with $u, v \in [\ 0, 2\pi]$, where r controls the thickness of the loops, and t gives the number of twists.

One interesting (although of little value in practical data analysis) feature of `wireframe()` is that it can draw parameterized surfaces. Such plots are created using the familiar formula `z ~ x * y`, but require `x`, `y`, and `z` to be all matrices with the same dimensions, representing coordinates of the parameterized surface evaluated over a grid of (u, v)-values. The following sequence of calls sets up the pieces required to create such matrices for the parameterization given above.

```
> kx <- function(u, v)
      cos(u) * (r + cos(u/2) * sin(t*v) - sin(u/2) * sin(2*t*v))
> ky <- function(u, v)
      sin(u) * (r + cos(u/2) * sin(t*v) - sin(u/2) * sin(2*t*v))
> kz <- function(u, v)
      sin(u/2) * sin(t*v) + cos(u/2) * sin(t*v)
> n <- 50
> u <- seq(0.3, 1.25, length = n) * 2 * pi
> v <- seq(0, 1, length = n) * 2 * pi
> um <- matrix(u, length(u), length(u))
> vm <- matrix(v, length(v), length(v), byrow = TRUE)
> r <- 2
> t <- 1
```

Figure 6.18 is now created with

```
> wireframe(kz(um, vm) ~ kx(um, vm) + ky(um, vm), shade = TRUE,
            screen = list(z = 170, x = -60),
            alpha = 0.75, panel.aspect = 0.6, aspect = c(1, 0.4))
```

6.4 Choosing a palette for false-color plots

Level plots encode a quantitative variable by using a color gradient (or grey levels) to represent numeric values. It is common to include a color key that maps the colors to the values they represent, but one should not expect to be able to use it to make accurate quantitative judgments. Rather, the primary usefulness of level plots is in judging patterns in the variability. A good choice of colors is often critical in how well a particular display serves this purpose.

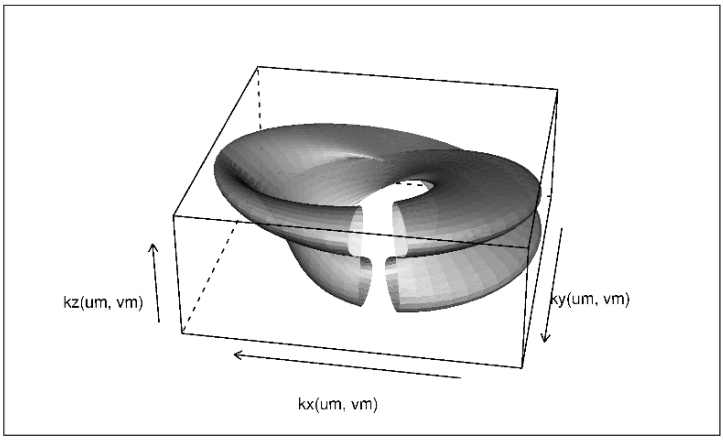

Figure 6.18. A shaded wireframe plot of the "figure 8" immersion of the Klein bottle, created using the parameterized form given in the text. The name comes from the interpretation of the object as a Möbius strip with the usual cross-section (a line segment) replaced by a double loop (like the number 8).

As a case in point, consider the `USAge.df` dataset available in the latticeExtra package, which records estimated population by age and sex in the United States between 1900 and 1979.

```
> data(USAge.df, package = "latticeExtra")
> str(USAge.df)
'data.frame':   12000 obs. of  4 variables:
 $ Age       : num  0 1 2 3 4 5 6 7 ...
 $ Sex       : Factor w/ 2 levels "Male","Female": 1 1 1 1 1 1 1 1 ...
 $ Year      : num  1900 1900 1900 1900 1900 1900 1900 1900 ...
 $ Population: num  0.919 0.928 0.932 0.932 0.928 0.921 0.911 0.899 ..
```

The dataset is large, with interesting local features. We look at some subsets of the data later in Chapter 10; here we consider a visualization of the full dataset using a level plot. In the following call, we use a gradient that is derived from a color palette designed by Cynthia Brewer (Harrower and Brewer, 2003; Neuwirth, 2007).

```
> library("RColorBrewer")
> brewer.div <-
      colorRampPalette(brewer.pal(11, "Spectral"),
                       interpolate = "spline")
> levelplot(Population ~ Year * Age | Sex, data = USAge.df,
            cuts = 199, col.regions = brewer.div(200),
            aspect = "iso")
```

The result, shown in Figure 6.19 along with other color plates, contains a small fluctuation around 1918 for males aged 22 or thereabouts. Unfortunately, neither the default black and white theme nor the default color theme will work well to highlight this feature. The palette used is by no means the only suitable choice; for example, the gradient produced by `terrain.colors()` also performs well in this case. The important point here is not that certain schemes are better than others, rather, it is that different color gradients emphasize different ranges of the data. One should keep this fact in mind when using color to encode numeric values.