Getting Started with Lattice Graphics

Deepayan Sarkar

`lattice` is an add-on package that implements Trellis graphics (originally developed for S and S-PLUS) in R. It is a powerful and elegant high-level data visualization system, with an emphasis on multivariate data, that is sufficient for typical graphics needs, and is also flexible enough to handle most nonstandard requirements. This lab covers the basics of `lattice` and gives pointers to further resources.

Some examples

To fix ideas, we start with a few simple examples. We use the `Chem97` dataset from the `mlmRev` package.

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

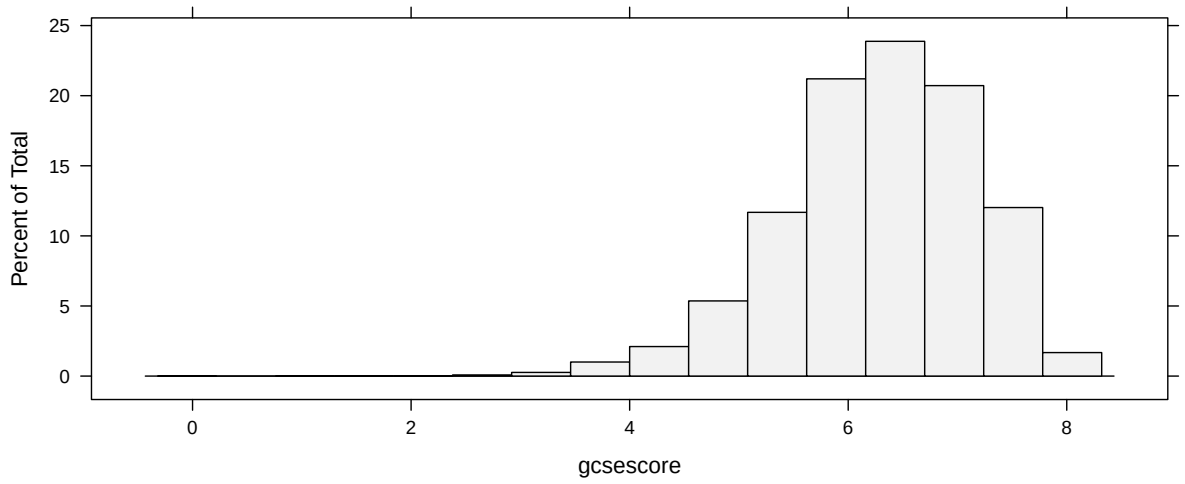
```
  lea school student score gender age gcsescore   gcsecnt
1   1      1       1     4      F   3     6.625 0.3393157
2   1      1       2    10      F  -3     7.625 1.3393157
3   1      1       3    10      F  -4     7.250 0.9643157
4   1      1       4    10      F  -2     7.500 1.2143157
5   1      1       5     8      F  -1     6.444 0.1583157
6   1      1       6    10      F   4     7.750 1.4643157
```

The dataset records information on students appearing in the 1997 A-level chemistry examination in Britain. We are only interested in the following variables:

- `score`: point score in the A-level exam, with six possible values (0, 2, 4, 6, 8).
- `gcsescore`: average score in GCSE exams. This is a continuous score that may be used as a predictor of the A-level score.
- `gender`: gender of the student.

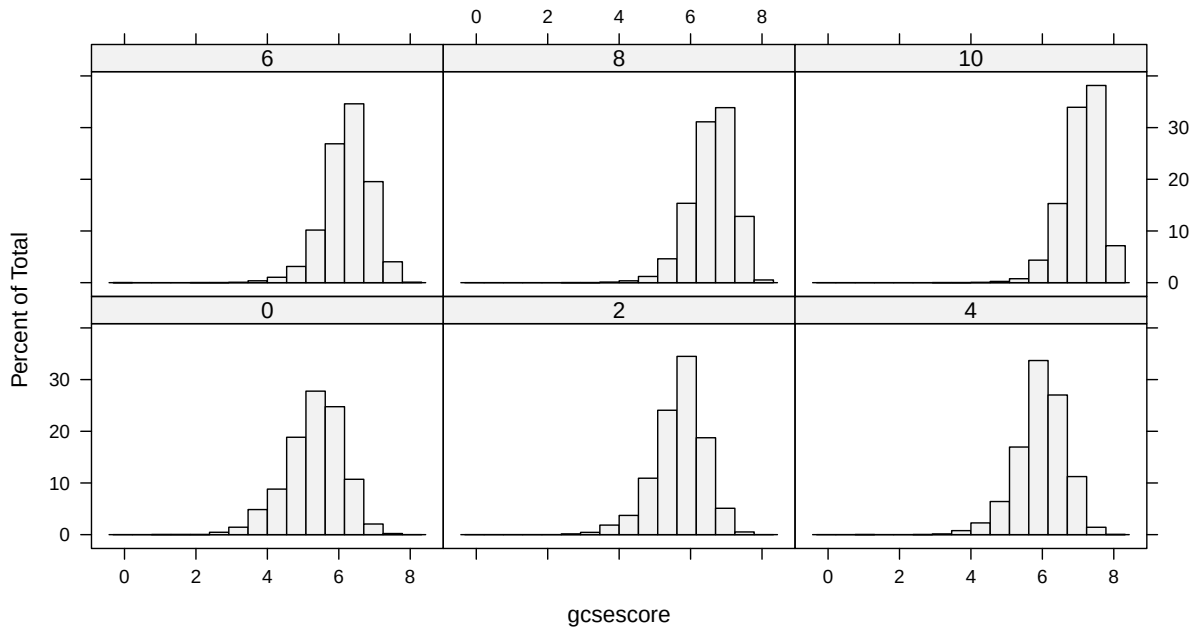
Using `lattice`, we can draw a histogram of all the `gcsescore` values using

```
> histogram(~ gcsescore, data = Chem97)
```



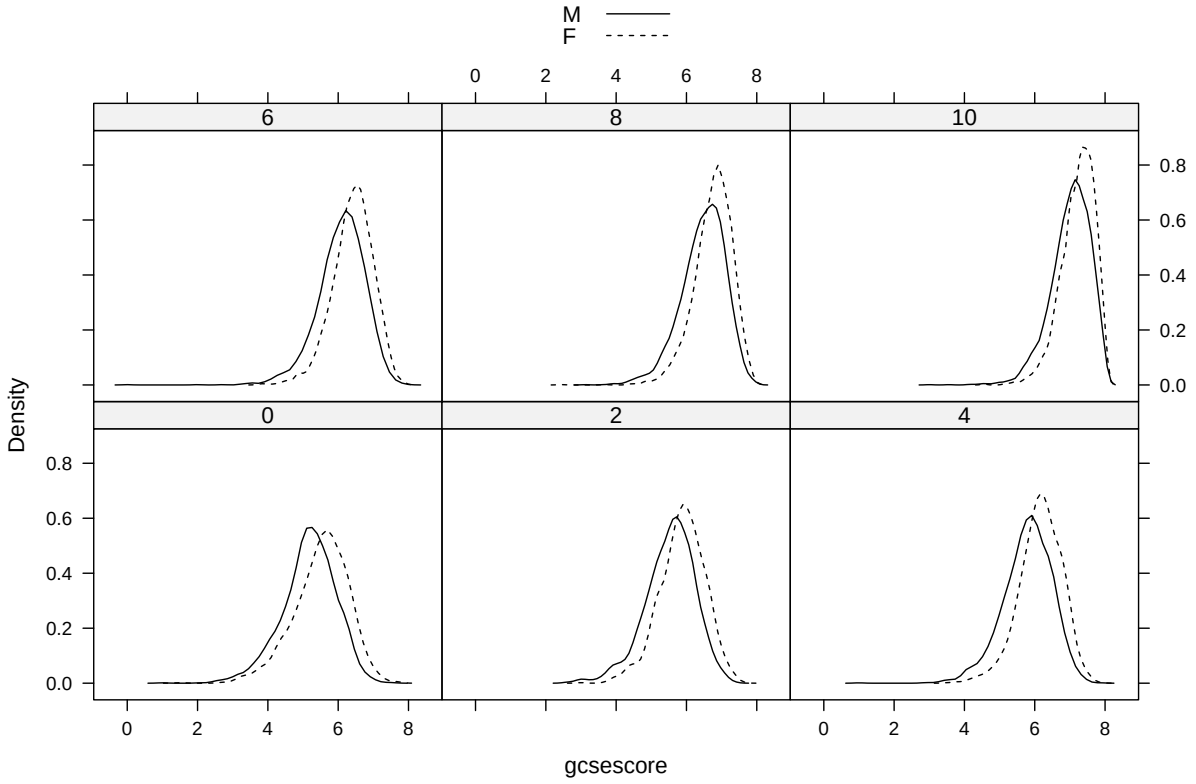
This plot shows a reasonably symmetric unimodal distribution, but is otherwise uninteresting. A more interesting display would be one where the distribution of `gcsescore` is *compared* across different subgroups, say those defined by the A-level exam score. This can be done using

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



More effective comparison is enabled by direct superposition. This is hard to do with conventional histograms, but easier using kernel density estimates. In the following example, we use the same subgroups as before in the different panels, but additionally subdivide the `gcsescore` values by `gender` within each panel.

```
> densityplot(~ gcsescore | factor(score), Chem97, groups = gender,
              plot.points = FALSE, auto.key = TRUE)
```



Exercise 1 *What happens if the extra arguments* `plot.points` *and* `auto.key` *are omitted? What happens if the inline call to* `factor()` *is omitted?*

The `plot.points` argument is described in the `?panel.densityplot` help page, and `auto.key` in `?xyplot`. Without the call to `factor()`, `score` is considered to be a numeric variable, and converted into a "shingle"; see `?shingle`.

Exercise 2 `lattice` *uses a system of customizable settings to derive default graphical parameters. Notice that the estimated densities for the two genders are distinguished by differing line types. Is this also true if you run the same commands in an interactive session? Which do you think is more effective?*

See `?trellis.device`.

Basic ideas

`lattice` provides a high-level system for statistical graphics that is independent of traditional R graphics.

- It is modeled on the Trellis suite in S-PLUS, and implements most of its features. In fact, `lattice` can be considered an implementation of the general principles of Trellis graphics (`?`).
- It uses the `grid` package (`?`) as the underlying implementation engine, and thus inherits many of its features by default.

Trellis displays are defined by the **type** of graphic and the **role** different variables play in it. Each display type is associated with a corresponding *high-level* function (`histogram`, `densityplot`, etc.). Possible roles depend on the type of display, but typical ones are

- primary variables: those that define the primary display (e.g., `gcsescore` in the previous examples).
- conditioning variables: divides data into subgroups, each of which are presented in a different panel (e.g., `score` in the last two examples).
- grouping variables: subgroups are contrasted within panels by superposing the corresponding displays (e.g., `gender` in the last example).

The following display types are available in `lattice`.

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

New high-level functions can be written to represent further visualization types; examples are `ecdfplot()` and `mapplot()` in the `latticeExtra` package.

Design goals

Visualization is an art, but it can benefit greatly from a systematic, scientific approach. In particular, **?** has shown that it is possible to come up with general rules that can be applied to design more effective graphs.

One of the primary goals of Trellis graphics is to provide tools that make it easy to apply these rules, so that the burden of compliance is shifted from the user to the software to the extent possible. Some obvious examples of such rules are:

- Use as much of the available space as possible
- Force direct comparsion by superposition (grouping) when possible
- Encourage comparison when juxtaposing (conditioning): use common axes, add common reference objects such as grids.

These design goals have some technical drawbacks; for example, non-wastage of space requires the complete display to be known when plotting begins, so, the incremental approach common in traditional R graphics (e.g., adding a main title after the main plot is finished) doesn't fit in. `lattice` deals with this using an object-based paradigm: plots are represented as regular R objects, incremental updates are performed by modifying such objects and re-plotting them.

While rules are useful, any serious graphics system must also be flexible. `lattice` is designed to be flexible, but obviously there is a trade-off between flexibility and ease of use for the more common tasks. `lattice` tries to achieve a balance using the following model:

- A display is made up of various elements
- The defaults are coordinated to provide meaningful results, but
- Each element can be controlled by the user independently of the others
- The main elements are:
  - the primary (panel) display
  - axis annotation
  - strip annotation (describing the conditioning process)
  - legends (typically describing the grouping process)

In each case, additional arguments to the high-level calls can be used to activate common variants, and full flexibility is allowed through arbitrary user-defined functions. This is particularly useful for controlling the primary display through panel functions.

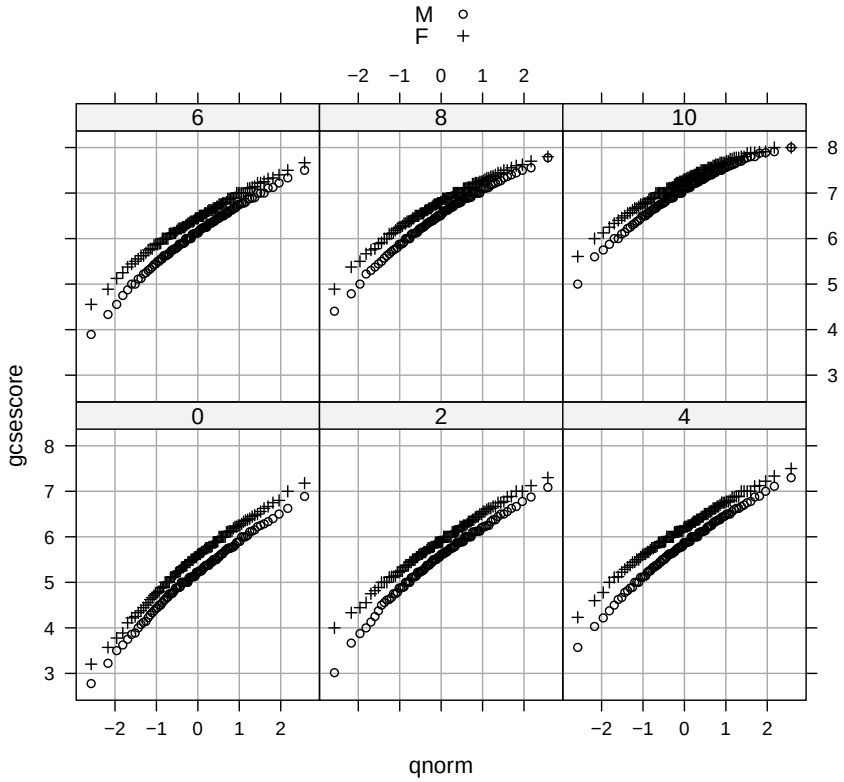
Most nontrivial use of `lattice` involves manipulation of one or more of these elements. Not all graphical designs segregate neatly into these elements; `lattice` may not be a good tool for such displays.

Common high-level functions

Visualizing univariate distributions

Several standard statistical graphics are intended to visualize the distribution of a continuous random variable. We have already seen histograms and density plots, which are both estimates of the probability density function. Another useful display is the normal Q-Q plot, which is related to the distribution function $F(x) = P(X \leq x)$. Normal Q-Q plots can be produced by the `lattice` function `qqmath()`.

```
> qqmath(~ gcsescore | factor(score), Chem97, groups = gender,
         f.value = ppoints(100), auto.key = TRUE,
         type = c("p", "g"), aspect = "xy")
```

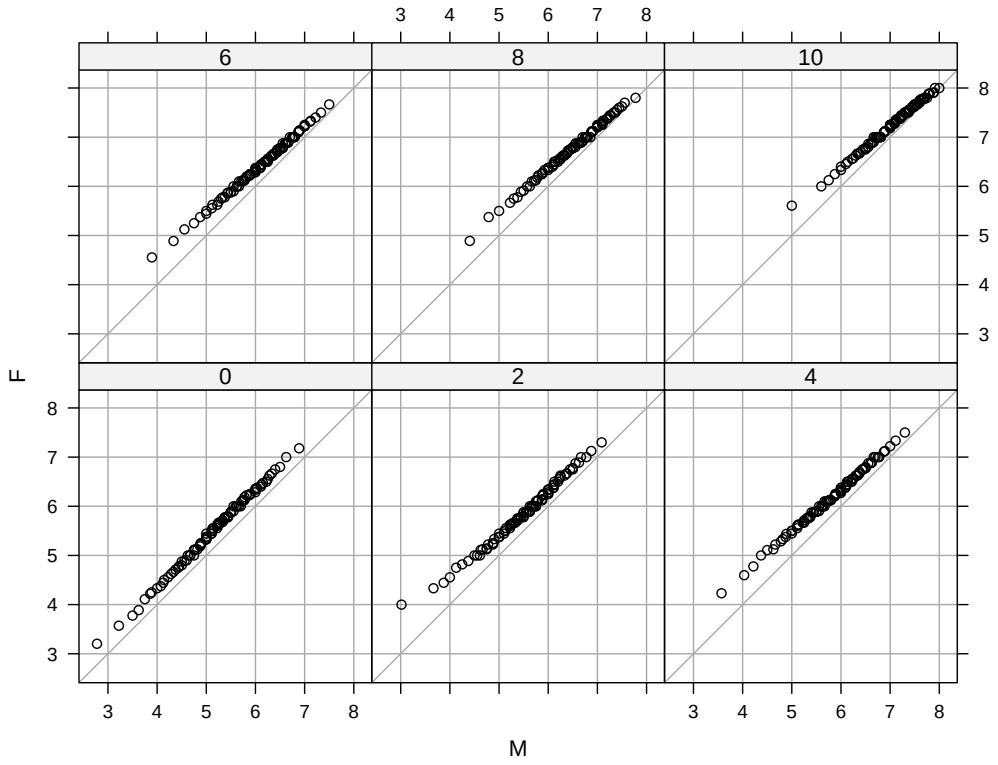


Normal Q-Q plots plot empirical quantiles of the data against quantiles of the normal distribution (or some other theoretical distribution). They can be regarded as an estimate of the distribution function F, with the probability axis transformed by the normal quantile function. They are designed to detect departures from normality; for a good fit, the points lie approximate along a straight line. In the plot above, the systematic convexity suggests that the distributions are left-skewed, and the change in slopes suggests changing variance.

The `type` argument adds a common reference grid to each panel that makes it easier to see the upward shift in `gcsescore` across panels. The `aspect` argument automatically computes an aspect ratio.

Two-sample Q-Q plots compare quantiles of two samples (rather than one sample and a theoretical distribution). They can be produced by the `lattice` function `qq()`, with a formula that has two primary variables. In the formula `y ~ x`, `y` needs to be a factor with two levels, and the samples compared are the subsets of `x` for the two levels of `y`. For example, we can compare the distributions of `gcsescore` for males and females, conditioning on A-level score, with

```
> qq(gender ~ gcsescore | factor(score), Chem97,
     f.value = ppoints(100), type = c("p", "g"), aspect = 1)
```

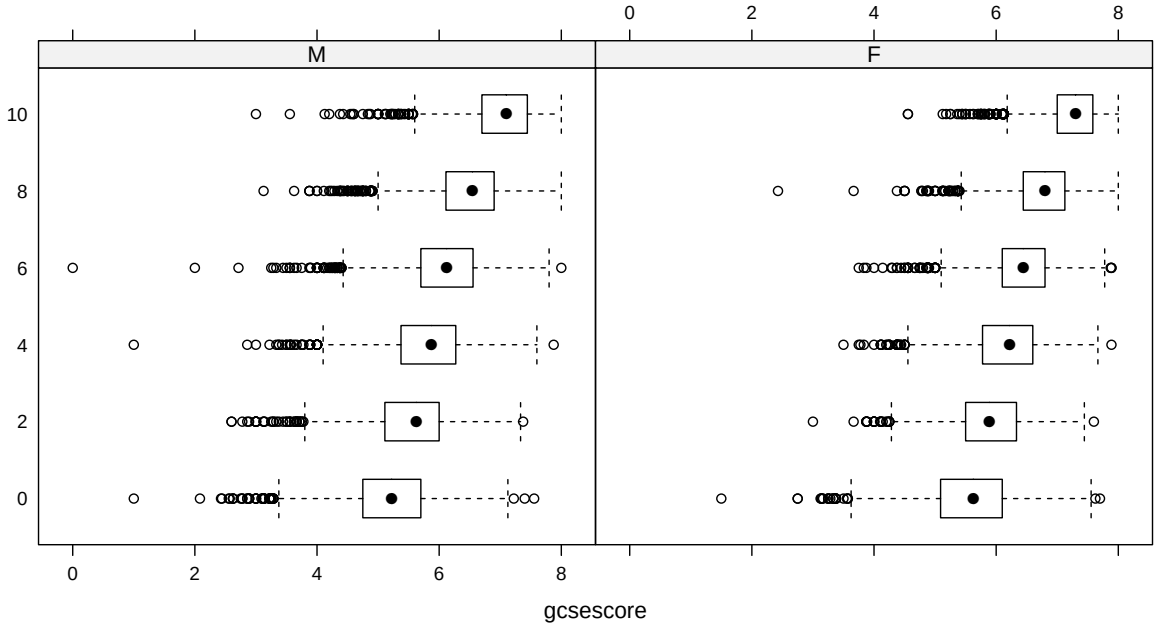


The plot suggests that females do better than males in the GCSE exam for a given A-level score (in other words, males tend to improve more from the GCSE exam to the A-level exam), and also have smaller variance (except in the first panel).

Two-sample Q-Q plots only allow comparison between two samples at a time. A well-known graphical design that allows comparison between an arbitrary number of samples is the comparative box-and-whisker plot. They are related to the Q-Q plot: the values compared are five "special" quantiles, the median, the first and third quartiles, and the extremes. More commonly, the extents of the "whiskers" are defined differently, and values outside plotted explicitly, so that heavier-than-normal tails tend to produce many points outside the extremes. See `?boxplot.stats` for details.

Box-and-whisker plots can be produced by the `lattice` function `bwplot()`.

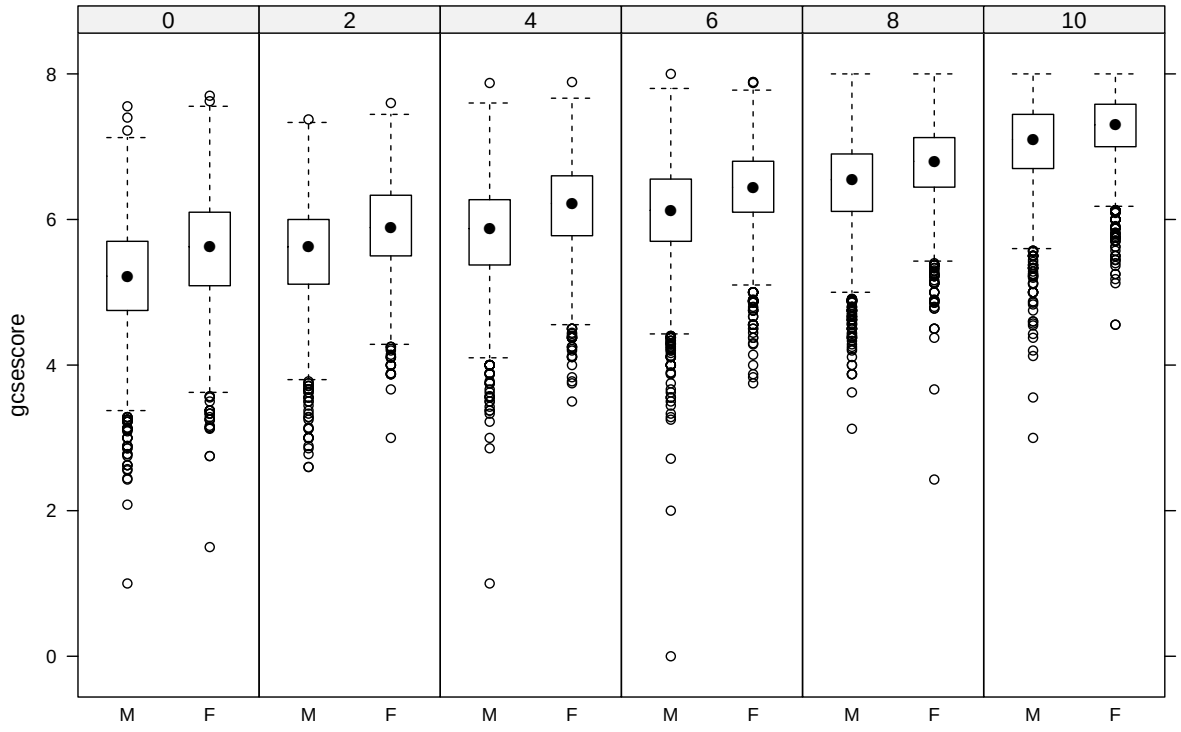
```
> bwplot(factor(score) ~ gcsescore | gender, Chem97)
```



The decreasing lengths of the boxes and whiskers suggest decreasing variance, and the large number of outliers on one side indicate heavier left tails (characteristic of a left-skewed distribution).

The same box-and-whisker plots can be displayed in a slightly different layout to emphasize a more subtle effect in the data: for example, the median `gcsescore` does not uniformly increase from left to right in the following plot, as one might have expected.

```
> bwplot(gcsescore ~ gender | factor(score), Chem97, layout = c(6, 1))
```



The `layout` argument controls the layout of panels in columns, rows, and pages (the default would not have been as useful in this example). Note that the box-and-whisker plots are now vertical, because of a switch in the order of variables in the formula.

Exercise 3 *All the plots we have seen suggest that the distribution of* `gcsescore` *is slightly skewed, and have unequal variances in the subgroups of interest. Using a Box–Cox transformation often helps in such situations. The* `boxcox()` *function in the* `MASS` *package can be used to find the "optimal" Box–Cox transformation, which in this case is approximate* 2.34*. Reproduce the previous plots replacing* `gcsescore` *by* `gcsescore^2.34`*. Would you say the transformation was successful?*

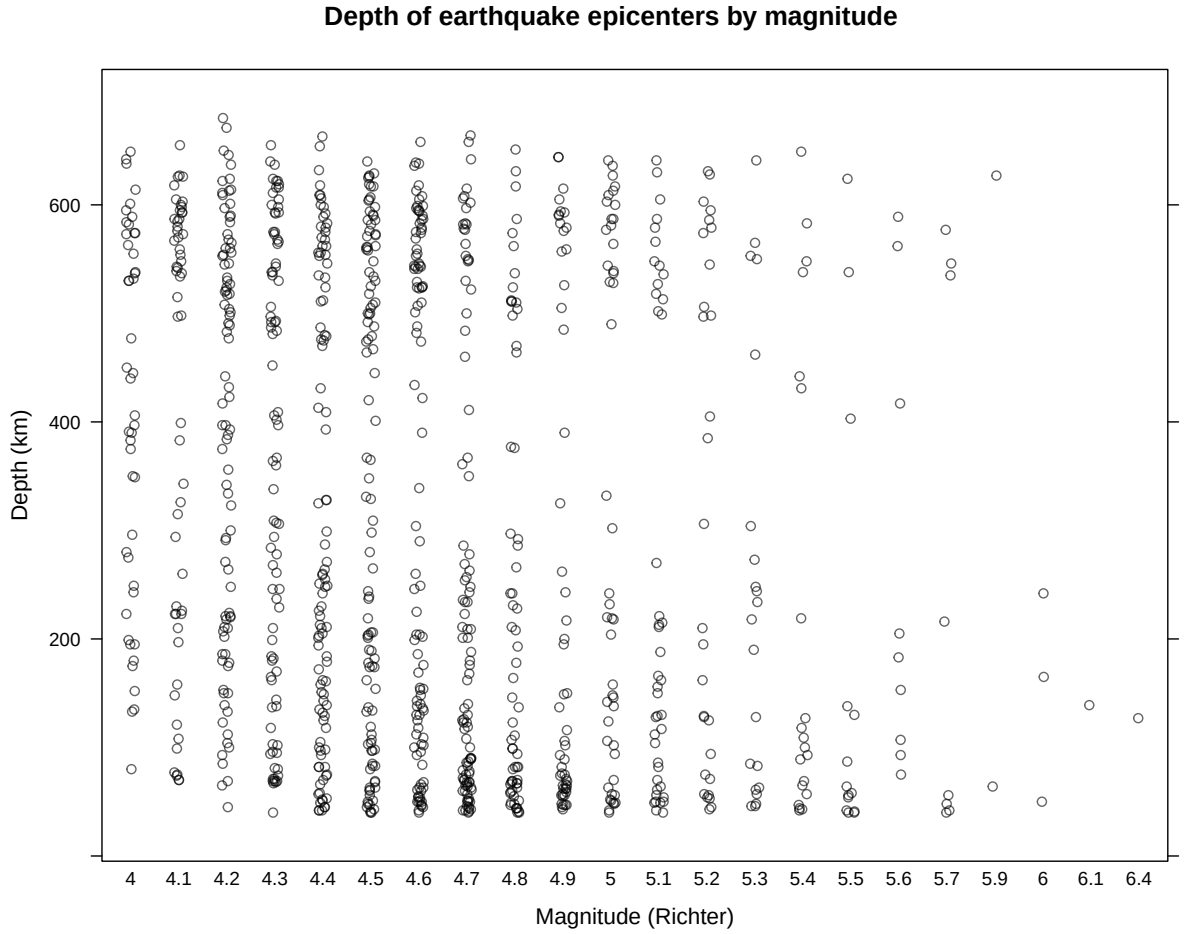
Exercise 4 *Not all tools are useful for all problems. Box-and-whisker plots, and to a lesser extent Q-Q plots, are mostly useful when the distributions are symmetric and unimodal, and can be misleading otherwise. For example, consider the display produced by*

```
> data(gvhd10, package = "latticeExtra")
> bwplot(Days ~ log(FSC.H), data = gvhd10)
```

What would you conclude about the distribution of `log(FSC.H)` *from this plot? Now draw kernel density plots of* `log(FSC.H)` *conditioning on* `Days`*. Would you reach the same conclusions as before?*

For small samples, summarizing is often unnecessary, and simply plotting all the data reveals interesting features of the distribution. The following example, which uses the `quakes` dataset, plots depths of earthquake epicenters by magnitude.

```
> stripplot(depth ~ factor(mag), data = quakes,
            jitter.data = TRUE, alpha = 0.6,
            main = "Depth of earthquake epicenters by magnitude",
            xlab = "Magnitude (Richter)",
            ylab = "Depth (km)")
```



This is known as a strip plot of a 1-D scatter plot. Note the use of jittering and partial transparency to alleviate potential overplotting. The arguments `xlab`, `ylab`, and `main` have been used to add informative labels; this is possible in all high-level `lattice` functions.

Visualizing tabular data

Tables form an important class of statistical data. Popular visualization methods designed for tables are bar charts and Cleveland dot plots.[1] For illustration, we use the `VADeaths` dataset, which gives death rates in the U.S. state of Virginia in 1941 among different population subgroups. `VADeaths` is a matrix.

```
> VADeaths
```

```
      Rural Male Rural Female Urban Male Urban Female
50-54       11.7          8.7       15.4          8.4
55-59       18.1         11.7       24.3         13.6
60-64       26.9         20.3       37.0         19.3
65-69       41.0         30.9       54.6         35.1
70-74       66.0         54.3       71.1         50.0
```

To use the `lattice` formula interface, we first need to convert it into a data frame.

```
> VADeathsDF <- as.data.frame.table(VADeaths, responseName = "Rate")
> VADeathsDF
```

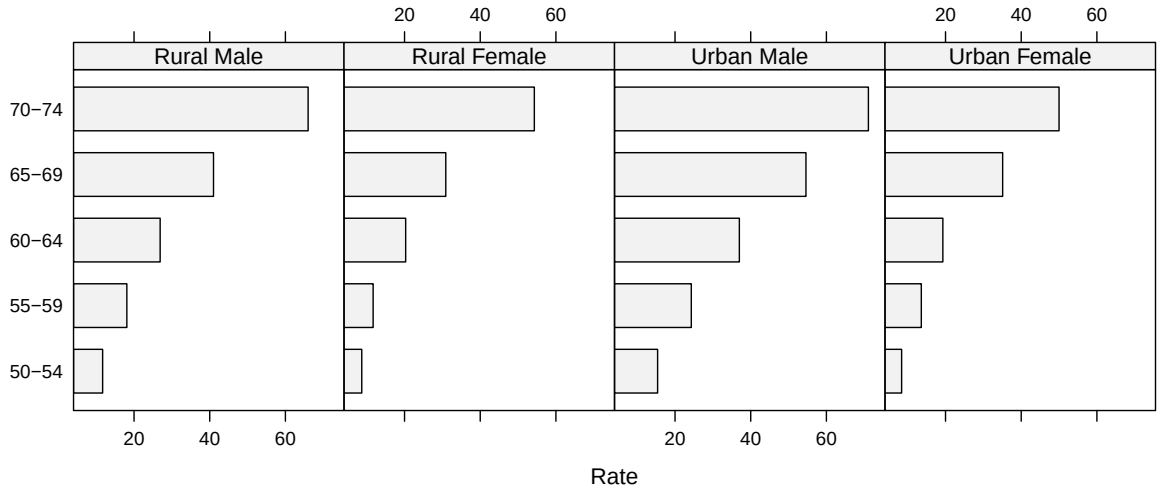
```
    Var1         Var2 Rate
1  50-54   Rural Male 11.7
2  55-59   Rural Male 18.1
3  60-64   Rural Male 26.9
4  65-69   Rural Male 41.0
5  70-74   Rural Male 66.0
6  50-54 Rural Female  8.7
7  55-59 Rural Female 11.7
8  60-64 Rural Female 20.3
9  65-69 Rural Female 30.9
10 70-74 Rural Female 54.3
11 50-54   Urban Male 15.4
12 55-59   Urban Male 24.3
13 60-64   Urban Male 37.0
14 65-69   Urban Male 54.6
15 70-74   Urban Male 71.1
16 50-54 Urban Female  8.4
17 55-59 Urban Female 13.6
18 60-64 Urban Female 19.3
19 65-69 Urban Female 35.1
20 70-74 Urban Female 50.0
```

Bar charts are produced by the `barchart()` function, and Cleveland dot plots by `dotplot()`. Both allow a formula of the form `y ~ x` (plus additional conditioning and grouping variables), where one of `x` and `y` should be a factor.

[1] Pie charts are also popular, but they have serious design flaws and should not be used. `lattice` does not have a high-level function that produces pie charts.

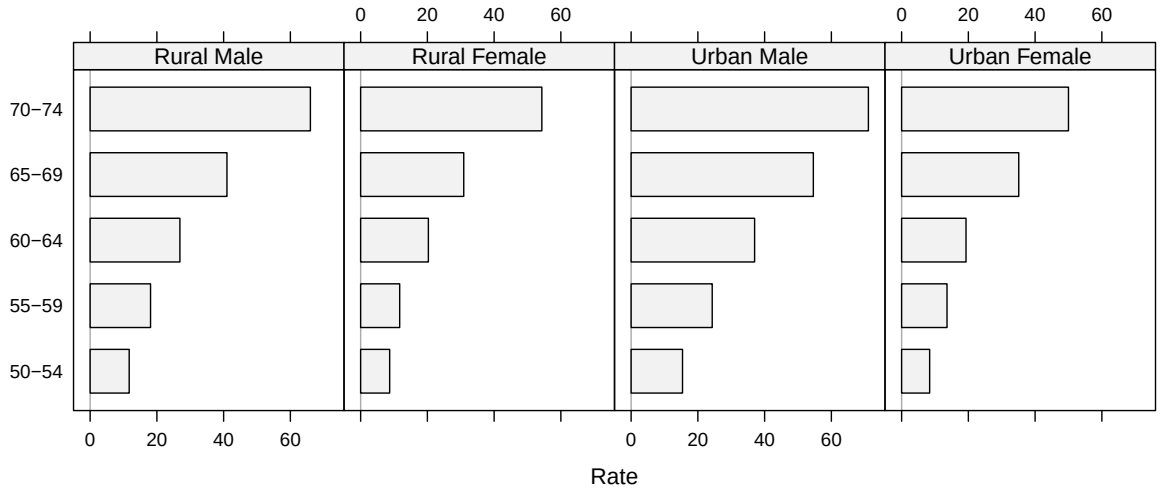
A bar chart of the `VADeathsDF` data is produced by

```
> barchart(Var1 ~ Rate | Var2, VADeathsDF, layout = c(4, 1))
```



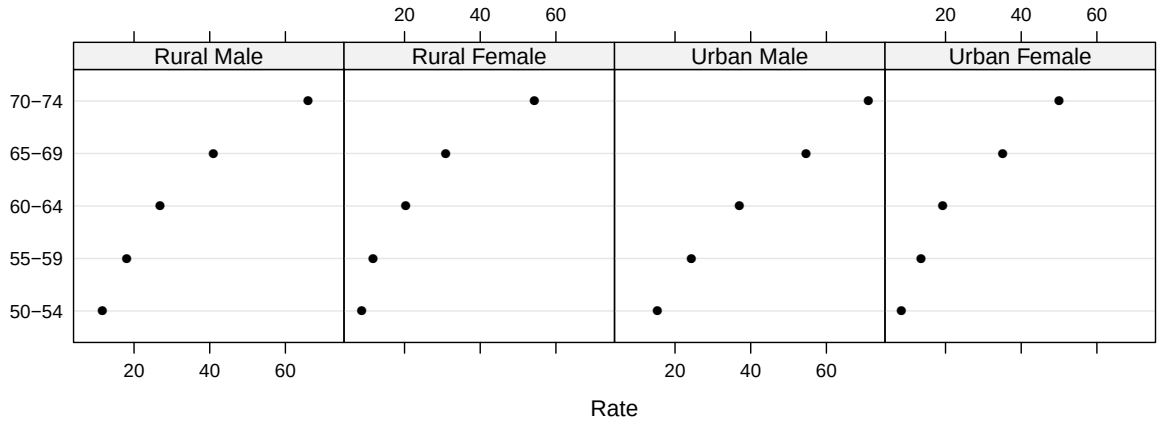
This plot is potentially misleading, because a strong visual effect in the plot is the comparison of the areas of the shaded bars, which do not mean anything. This problem can be addressed by making the areas proportional to the values they encode.

```
> barchart(Var1 ~ Rate | Var2, VADeathsDF, layout = c(4, 1), origin = 0)
```



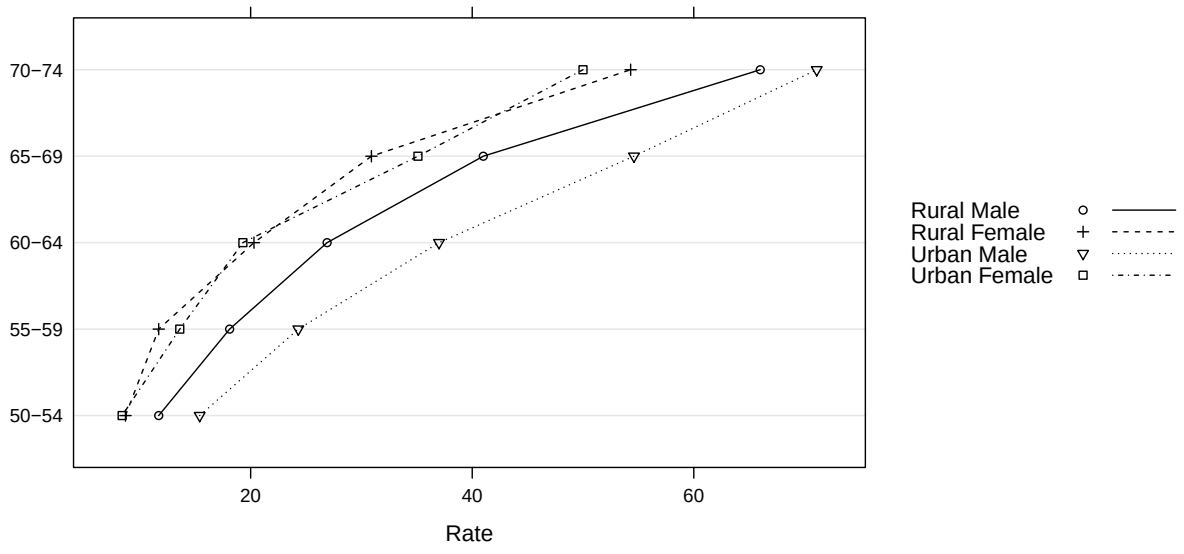
A better design is to altogether forego the bars, which distract from the primary comparison of the endpoint positions, and instead use a dot plot.

```
> dotplot(Var1 ~ Rate | Var2, VADeathsDF, layout = c(4, 1))
```



In this particular example, the display is more effective if we use `Var2` as a grouping variable, and join the points within each group.

```
> dotplot(Var1 ~ Rate, data = VADeathsDF, groups = Var2, type = "o",
          auto.key = list(space = "right", points = TRUE, lines = TRUE))
```

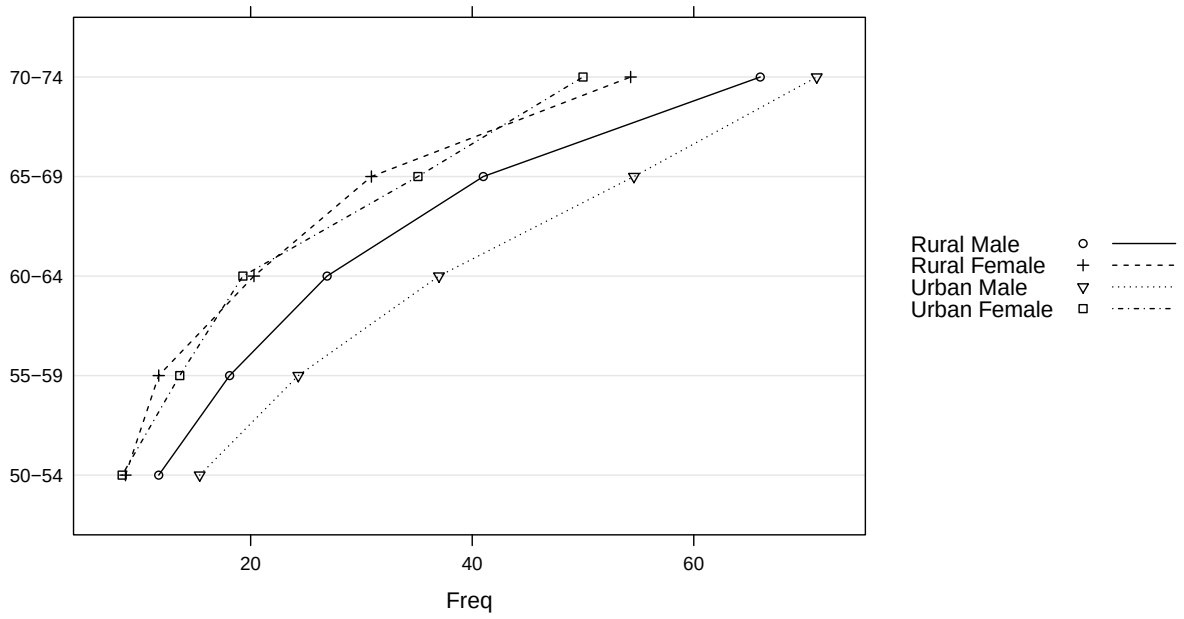


This plot clearly shows that the pattern of death rate by age is virtually identical for urban and rural females, with an increased rate for rural males, and a further increase for urban males. This interaction is difficult to see in the earlier plots.

Generic functions and methods

High-level `lattice` functions are actually generic functions, with specific methods doing the actual work. All the examples we have seen so far use the *"formula"* methods; that is, the method called when the first argument is a formula. Because `barchart()` and `dotplot()` are frequently used for multiway tables stored as arrays, `lattice` also includes suitable methods that bypass the conversion to a data frame that would be required otherwise. For example, an alternative to the last example is

```
> dotplot(VADeaths, type = "o",
          auto.key = list(points = TRUE, lines = TRUE, space = "right"))
```



Methods available for a particular generic can be listed using[2]

```
> methods(generic.function = "dotplot")
```

```
[1] dotplot.array*   dotplot.default* dotplot.formula*
[4] dotplot.matrix*  dotplot.numeric* dotplot.table*

   Non-visible functions are asterisked
```

The special features of the methods, if any, are described in their respective help pages; for example, `?dotplot.matrix` for the example above.

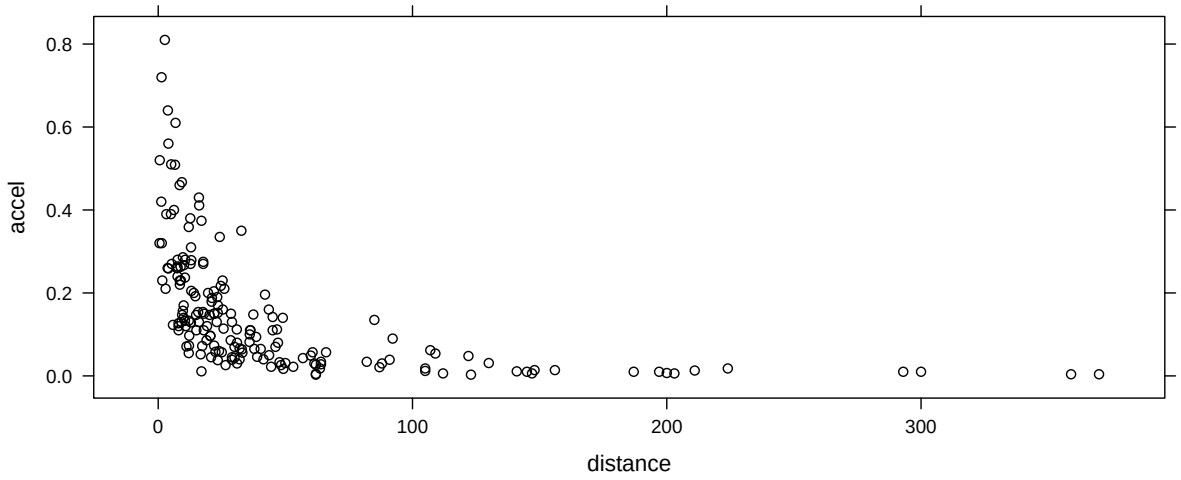
Exercise 5 *Reproduce the ungrouped dot plot using the "matrix" method.*

[2]This is only true for S3 generics and methods. `lattice` can be extended to use the S4 system as well, although we will not discuss such extensions here.

Scatter plots and extensions

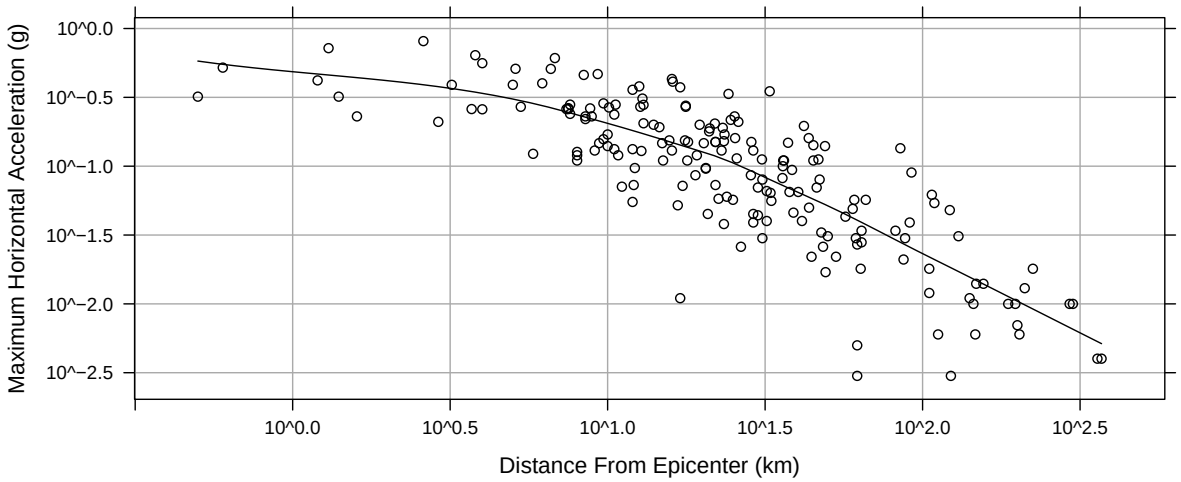
Scatter plots are commonly used for continuous bivariate data, as well as for time-series data. We use the `Earthquake` data, which contains measurements recorded at various seismometer locations for 23 large earthquakes in western North America between 1940 and 1980. Our first example plots the maximum horizontal acceleration measured against distance of the measuring station from the epicenter.

```
> data(Earthquake, package = "nlme")
> xyplot(accel ~ distance, data = Earthquake)
```



The plot shows patterns typical of a right skewed distribution, and can be improved by plotting the data on a log scale. It is common to add a reference grid and some sort of smooth; for example,

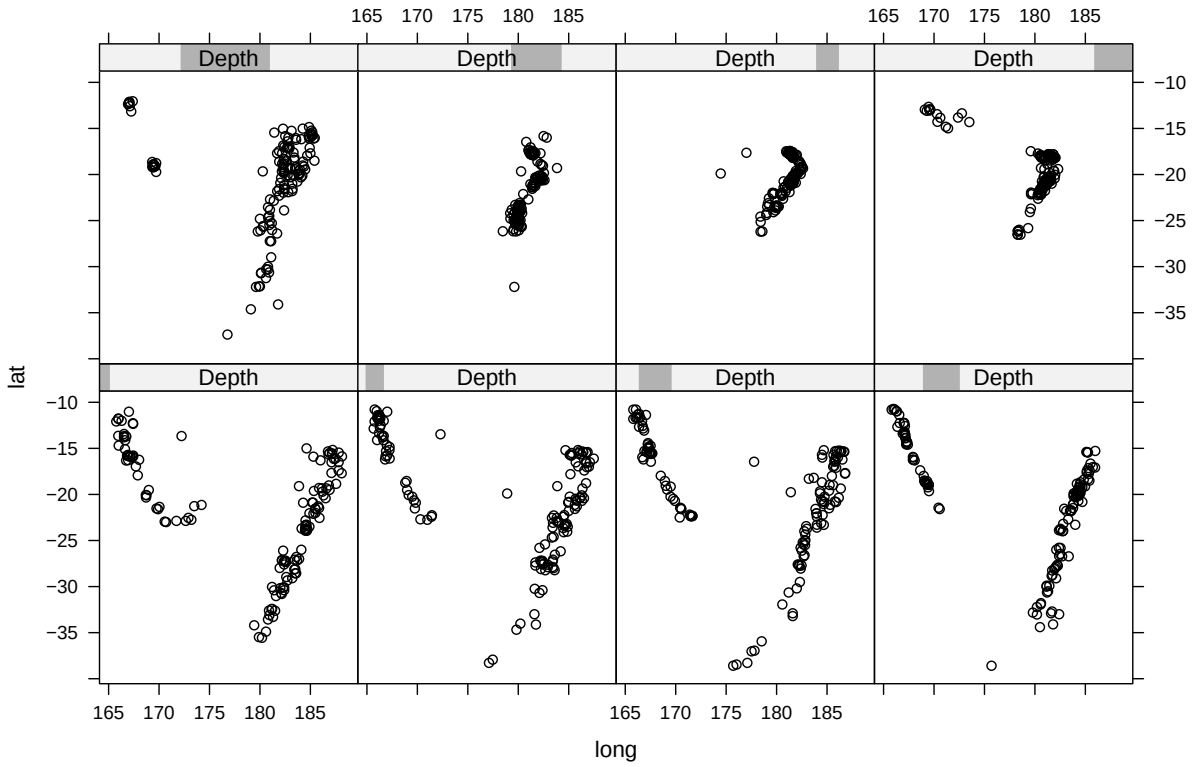
```
> xyplot(accel ~ distance, data = Earthquake, scales = list(log = TRUE),
         type = c("p", "g", "smooth"), xlab = "Distance From Epicenter (km)",
         ylab = "Maximum Horizontal Acceleration (g)")
```



Shingles

Conditioning by factors is possible with scatter plots as usual. It is also possible to condition on shingles, which are continuous analogues of factors, with levels defined by possibly overlapping intervals. Using the `quakes` dataset again, we can try to understand the three-dimensional distribution of earthquake epicenters by looking at a series of two-dimensional scatter plots.

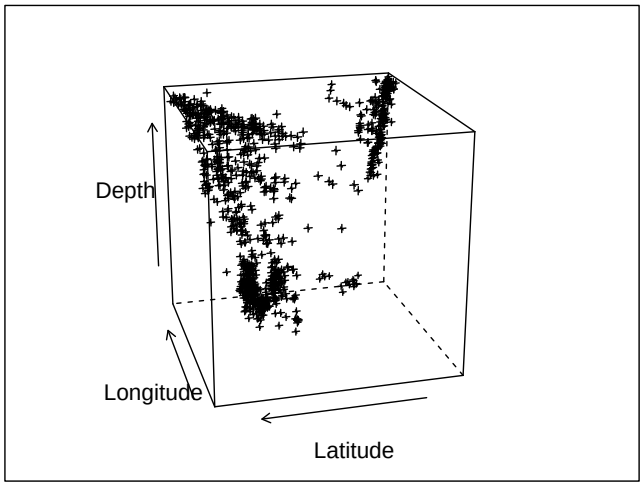
```
> Depth <- equal.count(quakes$depth, number=8, overlap=.1)
> summary(Depth)

Intervals:
    min   max count
1  39.5  63.5   138
2  60.5 102.5   138
3  97.5 175.5   138
4 161.5 249.5   142
5 242.5 460.5   138
6 421.5 543.5   137
7 537.5 590.5   140
8 586.5 680.5   137

Overlap between adjacent intervals:
[1] 16 14 19 15 14 15 15

> xyplot(lat ~ long | Depth, data = quakes)
```



Trivariate displays

Of course, for continuous trivariate data, it may be more effective to use a three-dimensional scatter plot.

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



Static three-dimensional scatter plots are not very useful because of the strong effect of "camera" direction. Unfortunately, `lattice` does not allow interactive manipulation of the viewing direction. Still, looking at a few such plots suggests that the epicenter locations are concentrated around two planes in three-dimensional space.

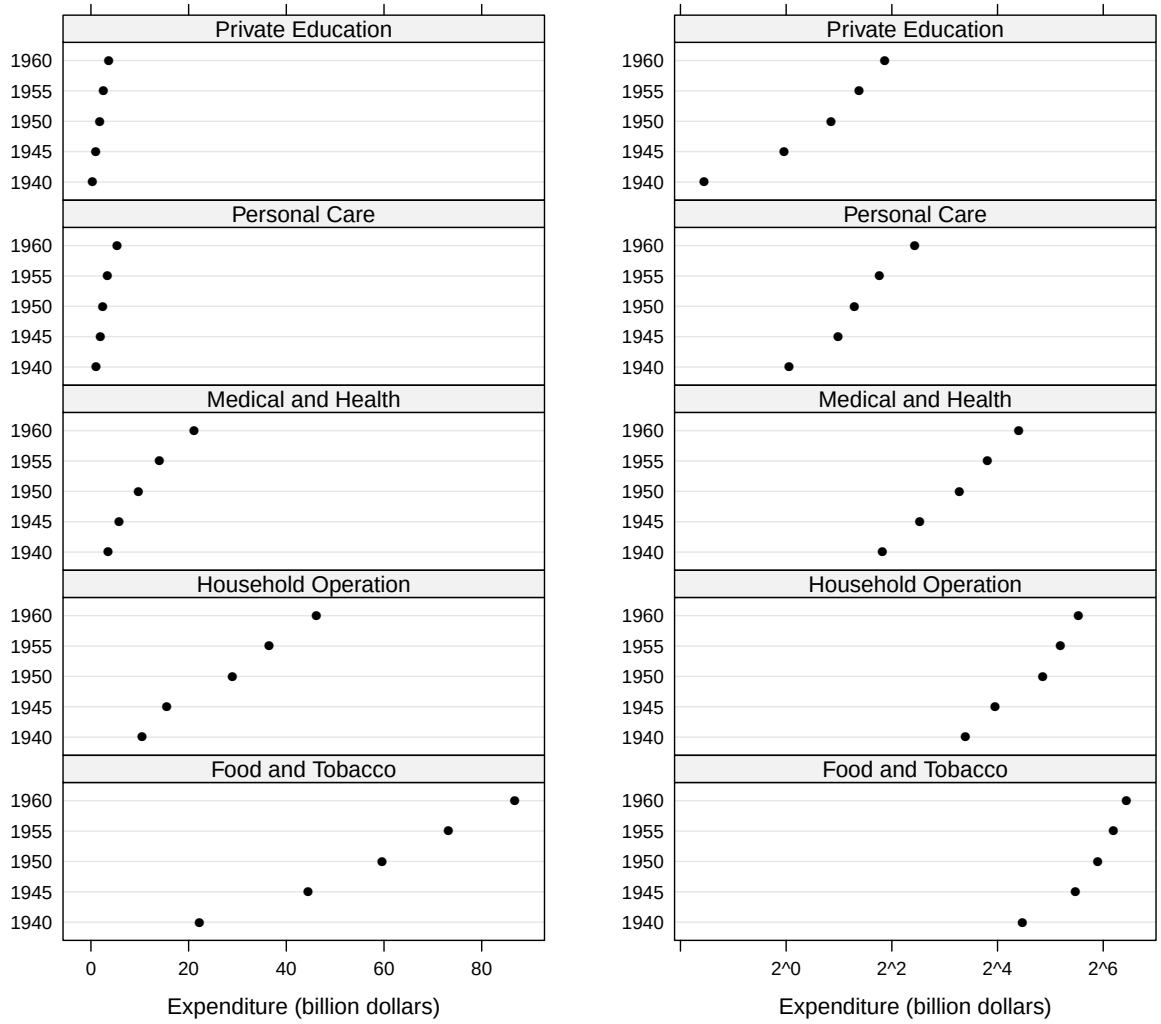
Other trivariate functions are `wireframe()` and `levelplot()`, which display data in the form of a three-dimensional surface. We will not explicitly discuss these and the other high-level functions in `lattice`, but examples can be found in their help pages.

Exercise 6 *Try changing viewing direction in the previous plot by modifying the* `screen` *argument.*

The *"trellis"* object

One important feature of `lattice` that makes it different from traditional R graphics is that high-level functions do not actually plot anything. Instead, they return an object of class *"trellis"*, that needs to be `print()`-ed or `plot()`-ed. R's automatic printing rule means that in most cases, the user does not see any difference in behaviour. Here is one example where we use optional arguments of the `plot()` method for *"trellis"* objects to display two plots side by side.

```
> dp.uspe <-
      dotplot(t(USPersonalExpenditure), groups = FALSE, layout = c(1, 5),
              xlab = "Expenditure (billion dollars)")
> dp.uspe.log <-
      dotplot(t(USPersonalExpenditure), groups = FALSE, layout = c(1, 5),
              scales = list(x = list(log = 2)),
              xlab = "Expenditure (billion dollars)")
> plot(dp.uspe,     split = c(1, 1, 2, 1))
> plot(dp.uspe.log, split = c(2, 1, 2, 1), newpage = FALSE)
```



Further resources

The material we have covered should be enough that the online documentation accompanying the `lattice` package should begin to make sense. Other useful material are

- The Lattice book, part of Springer's UseR! series. The book's website

  `http://lmdvr.r-forge.r-project.org`

  contains all figures and code from the book.

- The Trellis website from Bell Labs

  `http://netlib.bell-labs.com/cm/ms/departments/sia/project/trellis/`

  offers a wealth of material on Trellis that is largely applicable to `lattice` as well (of course, features unique to `lattice` are not covered)

Session information

- R version 2.14.0 Under development (unstable) (2011-05-07 r55801), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_IN`, `LC_NUMERIC=C`, `LC_TIME=en_IN`, `LC_COLLATE=en_IN`, `LC_MONETARY=en_IN`, `LC_MESSAGES=en_IN`, `LC_PAPER=C`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_IN`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: lattice 0.19-26
- Loaded via a namespace (and not attached): grid 2.14.0, tools 2.14.0