7

Graphics – base, lattice, ggplot2, rgl, googleVis. . .

Base Graphics (mostly 2-D):

Base graphics implements a "traditional" style of graphics

Functions `plot()`, `points()`, `lines()`, `text()`, `mtext()`, `axis()`,`identify()` etc. form a suite that plot points, lines, text, etc.

The function `plot()` accepts a `data` argument, while `lines()`, `points()` and `text()` do not.

Other Graphics

(i) lattice (trellis) graphics, using the *lattice* package,
(ii) *ggplot2*, implementing Wilkinson's *Grammar of Graphics*
(iii) For 3-D graphics (Section 7.5), note *rgl*, *misc3d* & *tkrplot*
(iv) *Motion Charts* (Section 7.5.1), show a scatterplot changing with movement forward or backward in time.

lattice and *ggplot2* are built on the low-level graphics package *grid*.

Note also various special types of graph. For example word clouds, as in the *wordcloud* package, list words with size proportional to frequency.

Consider first base graphics. Relative to *lattice* and to *ggplot2*, the more traditional style of base graphics is less consistent and less structured. Each system however has its own strengths and uses.

```
## DAAG has several datasets that will be required
library(DAAG, quietly=TRUE)
```

7.1 Base Graphics

The function `plot()` is the most basic of several functions that create an initial graph. Other functions can be used to add to an existing graph. Note in particular `points()` `lines()` and `text()`.

To see a variety of base (or traditional) graphics plots, enter

```
demo(graphics)
```

Press Enter to see each new graph.

7.1.1 `plot()` and allied base graphics functions

The following are alternative ways to plot `y` against `x` (obviously `x` and `y` must be the same length):

```
plot(y ~ x)    # Use a formula to specify the graph
plot(x, y)     # Horizontal ordinate, then vertical
```

Plot `height` vs `weight` –

```
## Older syntax:
with(women,
     plot(height, weight))
```

```
## Graphics formula:
plot(weight ~ height,
     data=women)
```

The following use the argument `data` to supply the name of a data frame whose column names appear in the graphics formula:

```
plot(distance ~ stretch, data=elasticband)
plot(ACT ~ year, data=austpop, type="l")
plot(ACT ~ year, data=austpop, type="b")
```

The `points()` function adds points, while `lines()` adds lines[1] to a plot. The `text()` function adds text at specified locations. The `mtext()` function places text in one of the margins. The `axis()` function gives fine control over axis ticks and labels.

[1] These functions differ only in the default setting for the parameter `type`. Explicitly setting `type = "p"` causes either function to plot points.

Here is a further possibility

```
with(austpop, plot(spline(year, ACT), type="l"))
  # Fit smooth curve through points
```

Adding text – an example

Here is a simple example (Figure 7.1) that uses the function `text()` to label the points on a plot. Data is from the dataset `primates` (*DAAG*). The first two lines of data are:

```
## Data (1st 2 lines)
head(primates, 2)
```

```
             Bodywt Brainwt
Potar monkey     10     115
Gorilla         207     406
```

Code for a simplified version of the plot is:

```
plot(Brainwt ~ Bodywt, xlim=c(0, 300),
     ylim=c(0,1500), data=primates, fg="gray",
     xlab="Body weight (kg)",
     ylab="Brain weight (g)")
# Specify xlim to allow room for the labels
with(primates,
     text(Brainwt ~ Bodywt, cex=0.8,
          labels=rownames(primates), pos=4))
# pos: pos=1 (below), 2 (left), 3 (above)
```

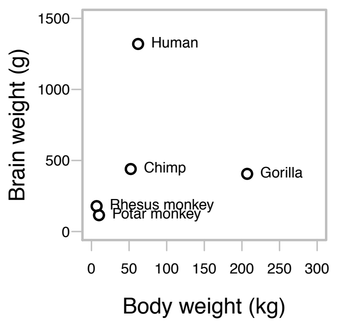


Figure 7.1: Plot of brain weight against body weight, for selected primates.

Identification and Location on the Figure Region

Draw the graph first, then call the required function.

- `identify()`, discussed in Subsection 1.5, labels points.
- `locator()` prints out the co-ordinates of points. Position the cursor at the location for which coordinates are required, and click the left mouse button.

Section 1.5 described how to terminate the plot, if the limit `n` is not reached first. For `locator()`, `n` is by default set to 500.

7.1.2 Fine control – Parameter settings

In most (not all) instances, parameters can be set either using `par()`, or in a call to a plotting function (`plot()`, `points()`, ...). Changes

To store existing settings for later restoration, proceed thus:

```
oldpar <- par(cex=1.25)
  # par(oldpar) to restore
```

made using `par()` remain in place until changed again, or until a new device is opened. If made in a call to a plotting function, the change applies only to that call.

Some of the more common settings are:

- Plotting symbols: `pch` (choice of symbol); `cex` ("character expansion")[2]; `col` (color).
- Lines: `lty` (line type); `lwd` (line width); `col` (color).
- Axis limits: `xlim`; `ylim`.
- Closeness of fit to the axis limits: `xaxs`, `yaxs`.[3] Specify `xaxs="i"` for an exact fit to the data limits.
- Axis annotation and labels: `cex.axis` (for axis annotation, independently of `cex`); `cex.labels` (for axis labels).
- Margins and positioning within margin:[4] `mar` (inner margins clockwise from bottom, out of box default `mar=c(5.1, 4.1, 4.1, 2.1)`); `oma` (outer margins, use when there are multiple plots on the one graphics page); positioning within margin: `mgp` (margin line for the axis title, axis labels, and axis line, default `mgp=c(3, 1, 0)`).
- Plot shape: `pty="s"` gives a square plot.[5] (default is `pty="m"`)
- Multiple graphs on the one graphics page: `par(mfrow=c(m,n))` gives an *m* rows by *n* columns layout.

For a 1 by 2 layout of plots; specify `par(mfrow=c(1,2))`. Subsection 3.2.1 has an example.

Type `help(par)` to get a (very extensive) complete list. Figure C.4 demonstrates some of the possibilities.

[2] Thus `par(cex=1.2)` increases plot symbol size 20% above the default.

[3] The default is `xaxs="r"`; *x*-axis limits are extended by 4% relative to data or `xlim` limits.

[4] Parameters such as `mar`, `mgp` and `oma` specify distances in 'lines' out from the relevant boundary of the figure region. Lines are in units of `mex`, where by default `mex=1`.

[5] This must be set using `par()`

7.1.3 Color and Opacity

The function `colors()` gives access to 657 different color names, some of them repeats of the same colour. The function `palette()` can be used to show or set colors that will by default be used for base graphics. Thus

- `palette()` lists the colors in the current palette;
- as an example, `palette(rainbow(6))` sets the current palette to a 6-color rainbow palette;
- `palette("default")` resets back to the default.

Run the following code to show the default palette, three sequential palettes from *grDevices*, a color ramp palette given by the function `colorRampPalette()`, and two quantitative palettes from the *RColorBrewer* package.

See `help(palette)` for palettes in the base R *grDevices* package.

```
## Load to run code for Supplementary Figure 1
library(RColorBrewer)   # Required for Set1 and Dark2 RColorBrewer palettes
```

```
colpal <- rev(list(
    "Default palette" = palette()[1:8],  cm.colors = cm.colors(12),
    terrain.colors = terrain.colors(12), heat.colors = heat.colors(12),
    blueRamp = colorRampPalette(c(blues9, "white"))(12),
    "Brewer-Set1" = brewer.pal(8, "Set1"),
    "Brewer-Dark2" = brewer.pal(8, "Dark2")))
palnam <- names(colpal)
plot(1, 1, xlim=c(0.5,12.5), ylim=c(0,length(palnam)+0.5), type="n",
     axes=FALSE, xlab="", ylab="")
for(i in 1:length(palnam)){
    len <- length(colpal[[i]])
    points(1:len, rep(i,len), pch=15, col=colpal[[i]], cex=5.5)
    legend(1, i+0.025, palnam[i], adj=0, box.col="white", bg="white",
           x.intersp=0, y.intersp=0, yjust=0)
```

Each of these palettes, except the default, allows variation in the number of colors, up to a maximum. The palettes available from *RColorBrewer* include other qualitative palettes, sequential palettes, and diverging (light in the middle; dark at the extremes) palettes. To see the full range of *RColorBrewer* possibilities, type:

Stretch the graphics window vertically (pull on an edge) so that rows do not overlap.

```
display.brewer.all()
```

Qualitative schemes that may be suited for use in plots are "Set1" with yellow (the 6th color out of nine) omitted, or "Dark2", or "Accent" with the 4th color (out of 8) omitted. To extract these, do for example:

While limited use of light colors is fine for coloring regions on a map, light colors do not show up well when coloring points on a graph.

```
Set1 <- brewer.pal(8, "Set1")[-6]
## Check out the palette
plot(1:7, pch=16, cex=2, col=Set1)
```

Opacity, and graphs with many points

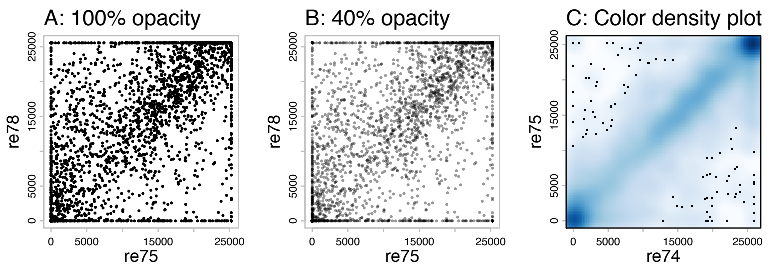


Figure 7.2: In Panel A, points are plotted with the 100% opacity, i.e., no transparency. In Panel B, `alpha=0.4`, i.e., 40% opacity. Panel C uses the function `smoothScatter()` to show a smoothed color density representation of the data.

An opacity of 0.4 has the effect that, for an isolated point, 60% of the white background shows through.

```
## Sample from the 15992 rows
dfsamp <- cps1[sample(nrow(cps1), 3000), ]
plot(re78 ~ re75, data=dfsamp, pch=20, cex=0.5,
     col="black", las=0, fg="gray")
mtext(side=3, line=0.5, "A: 100% opacity", adj=0)
plot(re78 ~ re75, data=dfsamp, pch=20, cex=0.5, las=0,
     col=adjustcolor("black", alpha=0.4), fg="gray")
mtext(side=3, line=0.5, "B: 40% opacity", adj=0)
```

```
blueRamp <- colorRampPalette(c("white", blues9))
with(dfsamp, smoothScatter(re75~re74, , fg="gray",
                             las=0, colramp=blueRamp))
mtext(side=3, line=0.5, "C: Color density plot",
      adj=0)
```

With alpha=0.4, two overlapping points have a combined opacity of 80%, so that 20% of the white background shows through. Three or more overlapping points appear as completely black.

Compare three plots shown in Figure 7.2. Points overlap to such an extent that Panel A, gives very limited information about the density of points. Panel B, where the color opacity is 40%, gives a better indication of variation in the density of points. Panel C uses the function smoothScatter() to provide a color density representation of the scatterplot. This is a more nuanced way to show the density of points.

The plots show a sample of 3000 of the points. Plotting all the points gives an incoveniently large graphics file, while not giving a more informative graph.

7.1.4 The shape of the graph sheet

Aspect ratio, i.e., the ratio of x-distance to y-distance, has a large say in what is visually obvious. Figures 7.3A and 7.3B show the same data: Features that are at an angle that is close to the horizontal or

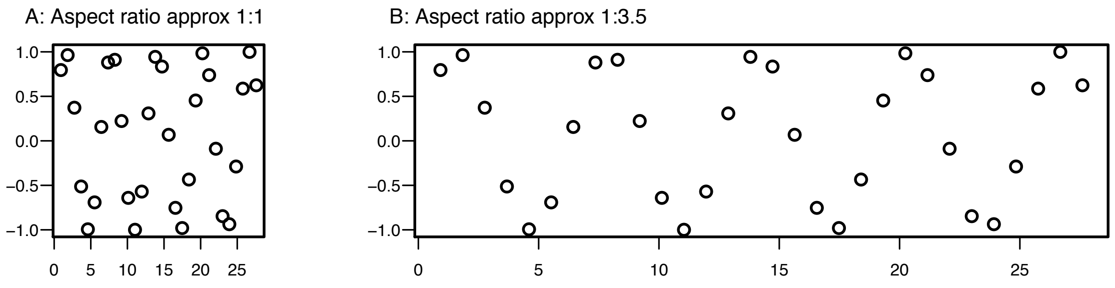


Figure 7.3: Figures A and B show the same data, but with widely different aspect ratios.

the vertical are hard to detect visually. Patterns of change or other features should, to be visually obvious, be offset by an angle of at least 20° from both the horizontal and the vertical.
For each of Figures 7.3A and 7.3B the code, after setting the dimensions of the figure page, is:

```
plot((1:30)*0.92, sin((1:30)*0.92),
     xlab="", ylab="")
```

The dimensions of the graphics display can be specified when a graphics window is opened. Once opened, the shape and size of a screen device can be changed by clicking and dragging on one corner.

The R for Windows functions win.graph() or x11() that set up the Windows screen take the parameters width (in inches), height (in inches) and pointsize (in 1/72 of an inch). The setting of pointsize (default =12) determines character heights. It is the relative sizes that matter for screen display or for incorporation into Word and similar programs.

7.1.5 Multiple plots on the one page

The parameter mfrow can be used to configure the graphics sheet so that subsequent plots appear row by row, one after the other in a rectangular layout, on the one page. The following presents four different transformations of data from the dataset Animals (*MASS*), in a two by two layout:

For a layout in which columns are filled before moving to a new row, use mfcol in place of mfrow.

```
## Supplementary figure 9.2
library(MASS)
oldpar <- par(pch=16, pty="s", mfrow=c(2,2))
with(Animals, {      # bracket several R statements
  plot(body, brain)
  plot(sqrt(body), sqrt(brain))
  plot(body^0.1, brain^0.1)
  plot(log(body), log(brain))
})                   # close both sets of brackets
par(oldpar)          # Restore former settings
```

A more flexible alternative is to use the graphics parameter fig to mark out the part of the graphics page on which the next graph will appear. The following marks out, successively, a plot region that occupies the upper 62% of the plot region, then the lower 38%.

```
par(fig = c(0, 1, 0.38, 1), mgp=c(3,0.5,0))
          # xleft, xright, ybottom, ytop
## Panel A
par(fig = c(0, 1, 0, 0.38), new=TRUE)
## Plot graph B
par(fig = c(0, 1, 0, 1))     # Restore settings
```

par(fig = c(0,1,0.38,1)) marks out a plot region that is the total width, starts 38% of the way up, and extends to the top. par(fig=c(0,1,0,0.38), new=TRUE) marks out the lower 38% of the page.

The effect of new=TRUE is, somewhat counter-intuitively, “assume a new page is already open; do not open a new page”.

7.1.6 Plots that show the distribution of data values

We discuss histograms, density plots, boxplots and normal probability plots. Normal probability plots are a specialised form of cumulative density plot.

Density plots are much preferable, for most purposes, to histograms. Both have limitations.

Histograms and density plots

The shapes of histograms depend on the placement of the breaks, as illustrated by Figure 7.4. The following code plots the histograms and superimposes the density plots.

```
par(mgp=c(3,0.5,0))
ftotlen <- subset(possum, sex=="f")[, "totlngth"]
## Left panel: breaks at 72.5, 77.5,..
hist(ftotlen, breaks = 72.5 + (0:5)*5, freq=FALSE,
     xlab="Total length", ylim=c(0,0.11),
     main ="A: Breaks at 72.5, 77.5,...")
```

The argument freq=FALSE gives a vertical scale that is the number of points per unit interval, i.e., it is the “density” estimate that is given by the upper bar of each rectangle. This is needed for the superposition of a density curve onto the histogram.

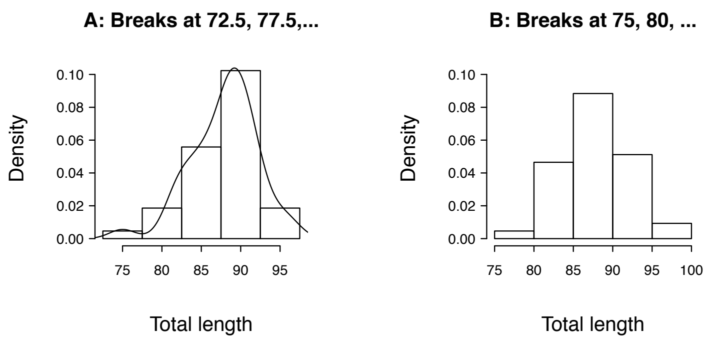


Figure 7.4: The two panels show the same data, but with a different choice of breakpoints.

```
## Now superimpose a density curve, as in Fig. 7.3
lines(density(ftotlen))
##
## Panel B: breaks at 75, 80, ...
hist(ftotlen, breaks = 75 + (0:5)*5, freq=FALSE,
     xlab="Total length", ylim=c(0,0.11),
     main="B: Breaks at 75, 80, ...")
```

The height of each rectangle of a histogram provides a crude density estimate. These estimates change in jumps, at breakpoints that are inevitably chosen somewhat arbitrarily. A smoothly changing density estimate, such as given by the superimposed density curves in the panels of Figure 7.4, makes better sense than an estimate that changes in jumps.

Unless samples are very large, the shape of both histograms and density plots will show large statistical variability. Density plots are helpful for showing the mode, i.e., the density maximum.

Neither histograms nor density plots are effective for checking normality. For that, use a normal probability plot.

The following gives a density plot, separately from the histograms that are shown in 7.4.

```
## Supplementary figure 9.3
with(subset(possum, sex=="f"),
     plot(density(totlngth), type="l"))
```

For use of density plots with data that have sharp lower and/or upper cutoff limits, it may be necessary to specify the x-axis limit or limits.[6] Use the parameters `from` and/or `to` for this purpose. This issue most commonly arises with a lower cutoff at 0.

[6] Thus, a failure time distribution will have a sharp cutoff at zero, which may also be the mode.

Boxplots

Boxplots use a small number of characteristics of a distribution to characterize it. Look up `help(boxplot)` for details. It can be insightful to add a "rug" that shows the individual values, by default along the horizontal axis (`side=1`). Figure 7.5 is an example. Code for the plot is:

```
## Code
with(subset(possum, sex=="f"),
     {boxplot(totlngth, horizontal=TRUE)
      rug(totlngth)} )
```

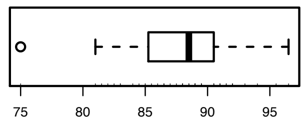


Figure 7.5: Distribution of lengths of female possums. The vertical bars along the x-axis (together making up a 'rug') show actual data values.

Normal probability plots

The function `qqnorm(y)` gives a normal probability plot of the values of `y`. In such a plot, data from a normal distribution will be scattered about a line. To calibrate the eye to recognise plots that indicate non-normal variation, it helps to compare the plot for the data in hand with several normal probability plots that use `rnorm()` to generate random values. Figure 7.6 is an example.

A point pattern that is not consistent with random deviation from a line indicates a non-normal distribution.

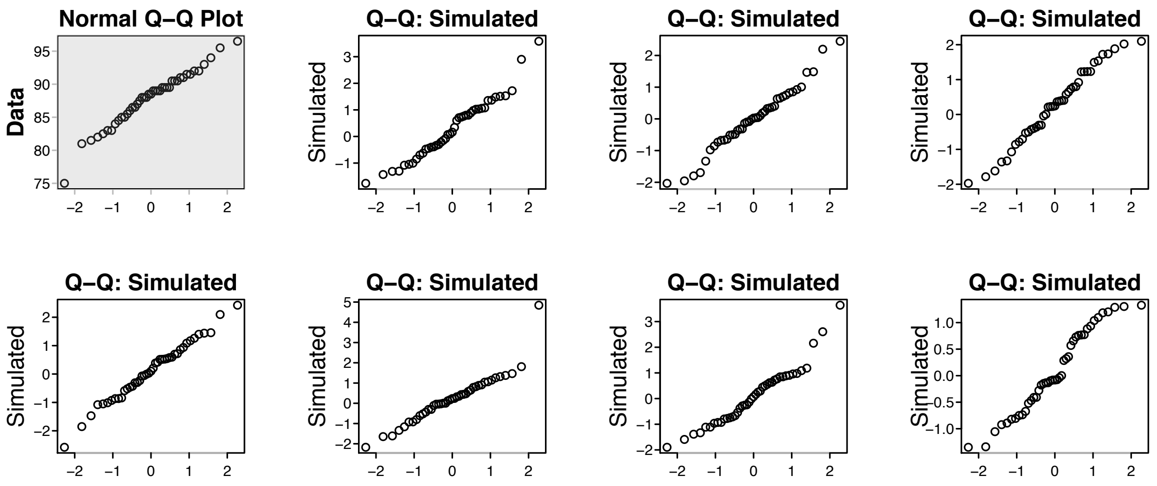


Figure 7.6: Normal probability plots. The top left panel shows the 43 lengths of female possums. Other panels are for independent normal random samples of size 43.

```
## Q-Q plot for the data (top left panel)
ftotlen <- subset(possum, sex == "f")[, "totlngth"]
qqnorm(ftotlen, xlab="",
       ylab=expression(bold("Data")))
## Code for a plot with random normal data
qqnorm(rnorm(43), xlab="", ylab="Simulated")
```

There is one unusually small value. Otherwise the points for the female possum lengths are as close to a straight line as in many of the plots for random normal data.

*7.1.7 *Plotting Text that Includes Technical Symbols*

The functions `expression()` and substitute() can be used to create mathematical expressions, for later evaluation or for printing onto a graph. For example, `expression(x^2)` will print, when supplied to `text()` or `mtext()` or another such function (this includes *lattice* and *ggplot2* functions), as x^2.

Axis labels can be expressions, in *lattice* and *ggplot2* as well as in base graphics. Tick labels can for example be vectors of expressions.

For purposes of adding text that includes mathematical and other technical symbols, the notion of expression is generalized, to allow "expressions" that it does not make sense to try to evaluate. For example, `expression("Temperature (" * degree * "C)")` prints as: Temperature (°C).

Items that are separated by an asterisk (*) are juxtaposed side by side. The initial text is followed by a degree symbol, and then by the final text.

The following indicate some of the possibilities:

- Letters such as a, b, c, x, ... are printed literally.

- `alpha` denotes the Greek letter α, while `Alpha` denotes the upper case symbol. Similarly for other Greek letters.
- `hat(x)` denotes $\hat{x}$.
- `italic(x)`, `bold(x)`, `bolditalic(x)`, and `plain(x)` have the obvious meaning.
- `frac(a,b)` denotes $\frac{a}{b}$.

Figure 7.7 demonstrates the use of an expression to provide *y*-axis labeling. The code is:

```
yl <- expression("Area = " * pi * r^~2)
plot(1:5, pi*(1:5)^2, xlab="Radius (r)", ylab=yl)
```

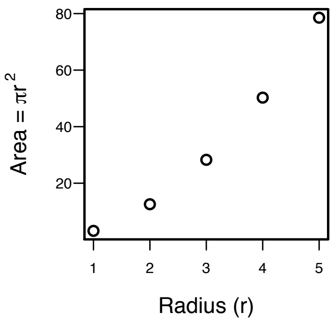


Figure 7.7: A mathematical expression is included as part of the *y*-axis label..

The tilde (~) in `r^~2` is used to insert a small space.

Use `substitute()` in place of `expression()` when symbols in the expression are to be replaced by values that will be provided at the time of forming the expression.

See `help(plotmath)` for further details of the conventions, and of the symbols that are available. Type `demo(plotmath)` to see a wide range of examples of what is possible.

7.2 *Lattice Graphics*

Lattice Graphics:	
Lattice	Lattice is a flavour of trellis graphics (the S-PLUS flavour was the original)
Lattice vs base	Lattice is more structured, automated and stylized. For standard purposes, much is automatic.
Lattice syntax	Lattice syntax is consistent and tightly regulated For lattice, graphics formulae are mandatory.

The *lattice* package is included in all R binary distributions that are available from a CRAN (Comprehensive R Archive Network) mirror. It implements a *trellis* style of graphics, as in the S-PLUS implementation of the S language. It is built on the *grid* low-level graphics system, described in Part II of Paul Murrell's *R Graphics*

To see some of the possibilities that lattice graphics offers, enter

```
demo(lattice)
```

Lattice (trellis) graphics functions allow the use of the layout on the page to reflect meaningful aspects of data structure. Groups can be readily distinguished within data, either using different colors and/or symbols and/or line types within panels or using different panels. Multiple columns of data can be plotted, either distinguished within panels or using different panels.

Functions that give styles of graph that are additional to those described here include `contourplot()`, `levelplot()`, `cloud()`, `wireframe()`, `parallel()`, `qqmath()` and `tmd()`.

Functions in the *latticeExtra* package further extend what is available.

In the discusssion that follows, there will be use of the layering abilities provided by functions in *latticeExtra*. Note that loading *latticeExtra* will at the same time load *lattice*, which *latticeExtra* has as a dependency.

These abilities, due to Felix Andrews, make it possible to build up lattice graphics objects layer by layer.

```
library(latticeExtra, quietly=TRUE)
```

7.2.1 Lattice graphics – basic ideas

Figure 7.8 was obtained using the lattice function `xyplot()`. In this simple case, the syntax closely matches that of the base graphics function `plot()`. Code is:

```
## On the command line: Create and print object
xyplot(Brainwt ~ Bodywt, data=primates)
```

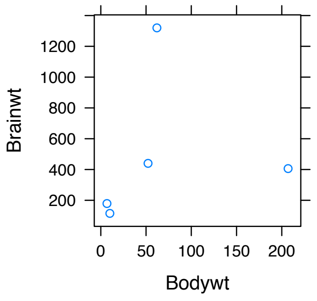


Figure 7.8: Use of lattice function `xyplot()` to give a graph.

Lattice graphics functions return graphics objects

Note an important difference between lattice and base graphics. Lattice graphics functions do not print graphs.[7] Instead they return trellis graphics objects. The graph appears when the object is printed (use `print()` or `plot()`). Sending the output from a *lattice* graphics function to the command line invokes `print()` and the graph is plotted, as was done for Figure 7.8.

[7] This applies also to *ggplot2*.

A `Brainwt` versus `Bodywt` scatterplot for the `primates` data, such as was given earlier, might alternatively have been obtained using the function the function `xyplot()` from the *lattice* package.

```
## Save the result as a trellis graphics object
# [For plot(), this is not possible.]
## Create trellis object
gph <- xyplot(Brainwt ~ Bodywt, data=primates)
## Print graph; a graphics device must now be open
print(gph)
```

The object `gph` need not be printed at this point. It can be kept for printing at some later time. Or it can be updated, using the function `update()`, and then printed, thus:

```
gph <- xyplot(Brainwt ~ Bodywt, data=primates)
gph2 <- update(gph, xlab="Body wt (kg)",
               ylab="Brain wt (g)")
print(gph2)  # Or it is enough to type 'gph2'
```

Inside a function or in a file that is sourced, `print()` must ordinarily be used to give a graph, thus:

```
print(xyplot(ACT ~ year, data=austpop))
```

The graph will however be printed if `xyplot(...)` is the final statement in a function that returns its result to the command line.

Addition of points, lines, text, . . .

For adding[8] to a plot that has been created using a *lattice* function, use `panel.points()`, `panel.text()`, and other such functions, as will be described in Subsection 7.2.8.

[8] Do not try to use `points()` and other such base graphics functions with lattice graphs.

Mechanisms for the control of a wide variety of stylistic features are best discussed in the context of multi-panel graphs, which we now consider.

7.2.2 *Panels of scatterplots*

Graphics functions in the *lattice* package, are designed to allow row by column layouts of panels. Different panels are for different subsets of the data. Additionally, points can be distinguished, within panels, according to some further grouping within the data.

The `ais` dataset (*DAAG*) has data from elite Australian athletes who trained at the Australian Institute of Sport. These included height, weight, and other morphometric measurements, as well as several types of blood cell counts. A breakdown of the total of 202 athletes by sex and sport gives:

See the help page for `ais` for details.

```
with(ais, table(sex,sport))
```

```
   sport
sex B_Ball Field Gym Netball Row Swim T_400m T_Sprnt Tennis W_Polo
  f     13     7   4      23  22    9     11       4      7      0
  m     12    12   0       0  15   13     18      11      4     17
```

Figure 7.9 demonstrates the use of `xyplot()`, for the rower and swimmer subset os the `ais` dataset. The two panels distinguish the two sports, while different plotting symbols (on a color device, different colors will be used) distinguish females from males.

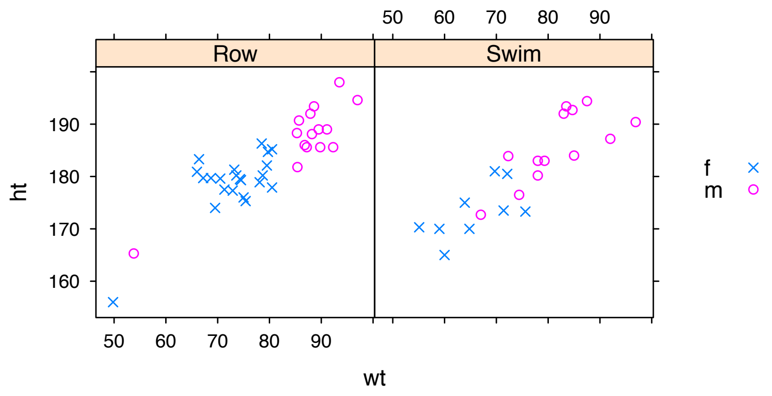


Figure 7.9: Height (`ht`) versus Weight (`wt`), for rowers (`Row`) and swimmers (`Swim`). Different plotting symbols are used to distinguish males from females.

Suitable code is:

```
xyplot(ht ~ wt | sport, groups=sex, data=ais,
       par.settings=simpleTheme(pch=c(4,1)),
       scales=list(tck=0.5),
       auto.key=list(space="right"),
       subset=sport%in%c("Row","Swim"))
```

Use `auto.key=list(columns=2)` to generate a simple key, with items side by side in two columns rather than stacked in a single column as is the default `columns=1`.

In the graphics formula `ht ~ wt | sport`, the vertical bar indicates that what follows, in this case `sport`, is a conditioning variable or factor. The graphical information is broken down by levels of the factor `sport`. The parameter `aspect` controls the ratio of dimensions in the *y* and *x* directions.

Subsection 7.2.3, which now follows, explains the use of the argument `par.settings`, and its call to `simpleTheme()`.

7.2.3 Setting stylistic features

The function `simpleTheme()` creates a "theme", i.e., a list of settings, that can be supplied via the argument `par.settings` in the graphics function call. Use of the argument `par.settings` to a lattice function makes the settings locally, for the specific graphics object that results.

The function `simpleTheme()` accepts arguments `col`, `alpha`, `cex`, `pch`, `lty`, `lwd`, `font`, `fill`, `border`, plus `col.points`, `col.line`, `alpha.points` and `alpha.line`. These allow separate control (of color and of opacity) for points and lines.

Settings that are not available using `simpleTheme()` can if required be added to the theme object that `simpleTheme()` returns. See Subsection 7.2.6 has details.

The function `trellis.device()` opens a new graphics device, with settings that have in mind the use of *lattice* functions. The function `trellis.par.set()` sets or changes stylistic features for the current device. Both these functions accept an argument `theme`.[9] Settings made by `trellis.device()` or `trellis.par.set()` will be over-written by any local settings that are stored as part of the graphics object.

[9] Simple variations on the default theme can be created by a call to `simpleTheme()`.

7.2.4 Groups within data, and/or columns in parallel

Table 7.1 shows selected rows from the data set `grog` (*DAAG* package). Each of three liquor products (drinks) has its own column. Rows are indexed by the factor `Country`.

	Beer	Wine	Spirit	Country	Year
1	5.24	2.86	1.81	Australia	1998
2	5.15	2.87	1.77	Australia	1999
. . . .					
9	4.57	3.11	2.15	Australia	2006
10	4.50	2.59	1.77	NewZealand	1998
11	4.28	2.65	1.64	NewZealand	1999
. . . .					
18	3.96	3.09	2.20	NewZealand	2006

Table 7.1: Apparent annual alcohol consumptiom values, obtained by dividing estimates of total available alcohol by number of persons aged 15 or more. These are based on Australian Bureau of Statistics and Statistics New Zealand figures.

Figure 7.10 is one of several possible displays that might be used to summarize the information in Table 7.1. It has been created by updating the following simplified code:

```
## Simple version of plot
grogplot <- xyplot(
              Beer+Spirit+Wine ~ Year | Country,
              data=grog, outer=FALSE,
              auto.key=list(space="right"))
```

Observe that:

- Use of `Beer+Spirit+Wine` gives plots for each of `Beer`, `Spirit` and `Wine`. The effect of `outer=FALSE` is that these appear in the same panel.

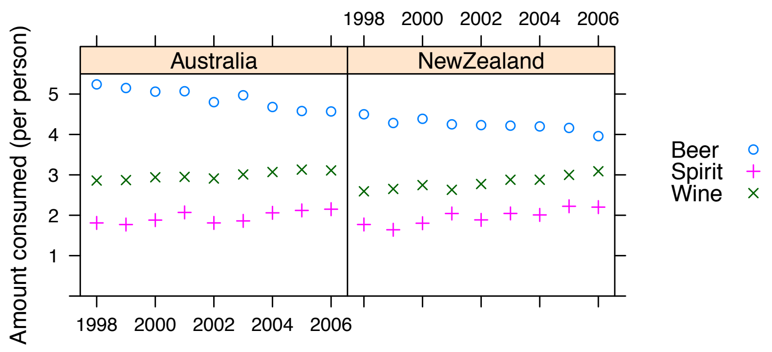


Figure 7.10: Australian and New Zealand apparent per person annual consumption (in liters) of the pure alcohol content of liquor products, for 1998 to 2006.

- Conditioning by country (| `Country`) gives separate panels for separate countries.

The following updates the object to give Figure 7.10:

```
## Update trellis object, then print
ylab <- "Amount consumed (per person)"
parset <- simpleTheme(pch=c(1,3,4))
finalplot <- update(grogplot, ylim=c(0,5.5),
                    xlab="", ylab=ylab,
                    par.settings=parset)
print(finalplot)
```

Notice the use of the function `simpleTheme()` to set up a “theme” that was used to control point and line settings.

Figure 7.10 used different symbols, in the one panel, to distinguish drinks, with different countries in different panels. For separate panels for the three liquor products (different levels of `Country` can then use the same panel), specify `outer=TRUE`:

```
xyplot(Beer+Spirit+Wine ~ Year,
       groups=Country, outer=TRUE,
       data=grog, auto.key=list(columns=2) )
```

Where plots are superposed in the one panel and, e.g., regression lines or smooth curves are fitted, this is done separately for each different set of points. Different colors, and/or by different symbols and/or line styles, can be used to make the necessary distinctions.

Here is a summary:

Break data down a/c to levels of the factor `Country`:

Overplot (a single panel):	Separate panels:
`Beer ~ Year, groups=Country`	`Beer ~ Year	Country`

Plot columns in parallel, as in `Beer+Wine+Spirit ~ Year`:

Overplot (a single panel):	Separate panels:
`outer=FALSE`	`outer=TRUE`

7.2.5 *Keys – `auto.key`, `key` and `legend`*

The argument `auto.key=TRUE` gives a basic key. If not otherwise

The argument `auto.key` sets up a call `key=simpleKey()`. If necessary, use `legend=NULL` when updating, to remove an existing key and allow the addition of a new key.

specified, colors, plotting symbols, and line type use the current settings for the device. The argument `text` has `levels(groups)` as its default. that identifies colors, plotting symbols and names for the groups. For greater flexibility, `auto.key` can be a list. Settings that are often useful are:

- `points`, `lines`: in each case set to `TRUE` or `FALSE`.
- `columns`: number of columns of keys.
- `x` and `y`, which are coordinates for the whole display area. Use with `corner` set to one of `c(0,0)`, `c(1,0)`, `c(1,1)` and `c(0,1)`.
- `space`: one of `"top"`, `"bottom"`, `"left"`, `"right"`.

`c(0,0)` is the bottom left corner of the legend, etc.

*Use of `textGrob()` to add legends

The function `textGrob()` (*grid*) creates a text object which can then be supplied to the lattice function. This mechanism for supplying legends can be used when multiple legends are required.

The following code adds an initial legend, as in Figure 7.11:

```
plotnam <- "Stripplot of cuckoo data"
stripplot(species ~ length, xlab="", data=cuckoos,
  legend=list(top=list(fun=grid::textGrob,
                       args=list(label=plotnam,
                                 x=0))))
# x=0 is equivalent to x=unit(0,"npc")
# npc units are on a scale from 0 to 1
```

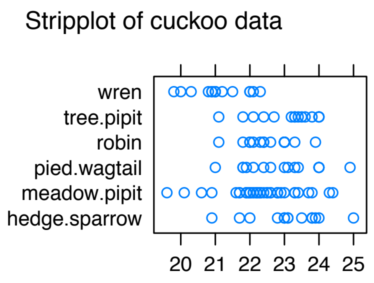


Figure 7.11: The argument `legend` has been used to add text, supplied as a 'grob'.. Here, it would be easier to use of the argument `main`.

Additional legends are supplied by adding further list elements, for example a list element `bottom` as well as a list element `top`.

7.2.6 *Lattice settings – further notes*

In general, use `themes` to make point, line and fill color settings. Use the `scales` argument, in the call to the lattice function, for axes, tick marks, and tick labels.

For changes that go beyond what `simpleTheme()` allows, first identify the names under which settings are stored. Type:

```
> names(trellis.par.get())
[1] "fontsize"          "background"       "clip"
. . .
[28] "par.sub.text"
```

For a visual display that shows default settings for points, lines and fill color, enter:

```
trellis.device(color=FALSE)
show.settings()
trellis.device(color=TRUE)
show.settings()
```

The following sets the fontsize, separately for `text` and `points`:

```
trellis.par.set(fontsize = list(text = 7,
                                points = 4))
```

Parameters that affect axes, tick marks, and axis labels

These are manipulated by use of the `scales` argument to the lattice function. The code for Figure 7.12 provides an example.

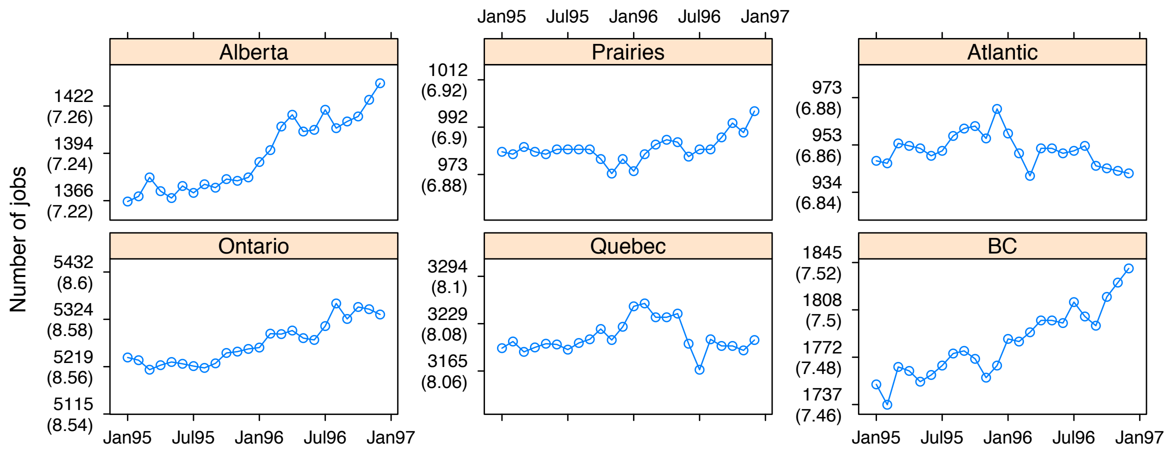


Figure 7.12: Jobs growth in Canadian provinces, between January 1995 and December 1996.

The following gives a basic graph, which will then be updated:

```
## 1. Create a basic version of the graphics object
jobsB.xyplot <-
  xyplot(Ontario+Quebec+BC+Alberta+Prairies+Atlantic ~ Date,
         data=jobs, type="b", layout=c(3,2), outer=TRUE,
         ylab="Number of jobs",
         scales=list(y=list(relation="sliced", log=TRUE)))
```

Now make several enhancements:

- Change the y-axis labels to show number of jobs, with log(number) in parentheses underneath.
- Use dates of the form `Jan95` to label the x-axis.[10]
- Reduce tick marks in length (`tck=0.6`, i.e., 60% of the default).
- The argument `between=list(x=0.5, y=0.5)` adds horizontal and vertical space between the panels.[11]

[10] Refer back to Subsection 4.3.9.

[11] This avoids overlap of tick labels.

```
## 2. Code for the enhancements to jobsB.xyplot
ylabpos <- exp(pretty(log(unlist(jobs[,-7])), 100))
ylabels <- paste0(round(ylabpos),"\n(", log(ylabpos), ")")
## Create a date object 'startofmonth'; use instead of 'Date'
atdates <- seq(from=95, by=0.5, length=5)
datelabs <- format(seq(from=as.Date("1Jan1995", format="%d%b%Y"),
                       by="6 month", length=5), "%b%y")
update(jobsB.xyplot, xlab="", between=list(x=0.5, y=0.5),
       scales=list(x=list(at=atdates, labels=datelabs),
                   y=list(at=ylabpos, labels=ylabels), tck=0.6) )
```

7.2.7 *Lattice plots that show distributions*

Stripplots, dotplots and boxplots

Because the syntax for `stripplot()` and `boxplot()` are very similar, we demonstrate suitable code side by side. Figure 7.13 summarizes cuckoo egg length data, from the dataset `cuckoos` from *DAAG*:

Differences between `dotplot()` and `stripplot()` are mainly cosmetic.

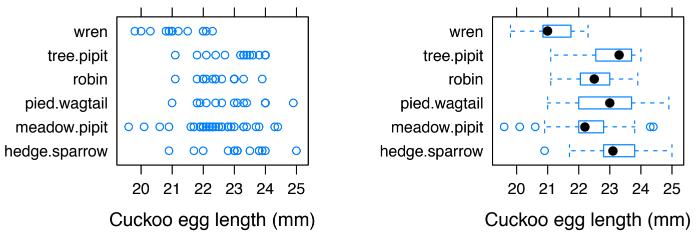


Figure 7.13: A stripplot and a dotplot appear side by side.

```
stripplot(species ~ length, data=cuckoos,
          xlab="Cuckoo egg length (mm)")
bwplot(species ~ length, data=cuckoos,
       xlab="Cuckoo egg length (mm)")
```

For slightly improved labeling, precede the code with:

```
levels(cuckoos$species) <-
  sub(".", " ",
    levels(cuckoos$species),
    fixed=TRUE)
```

The `aspect` argument determines the ratio of distance in the y-direction to distance in the x-direction.

Lattice style density plots

Here is a density plot (Figure 7.14), for data from the `possum` data set (*DAAG*), that compares `sexes` and `Vic/other` populations.

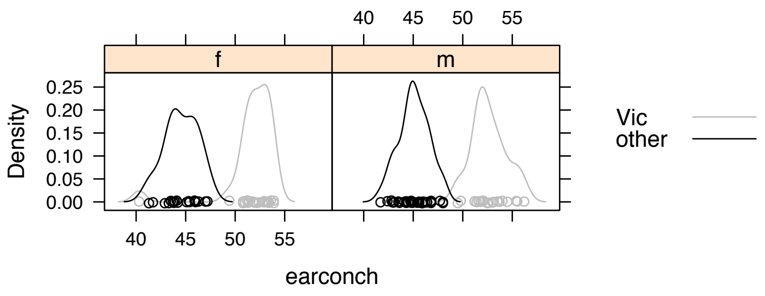


Figure 7.14: Lattice style density plot comparing possum earconch measurements, separately for males and females, between Victorian and other populations. Observe that the scatter of data values is shown along the horizontal axis.

```
## Code
colset <- c("gray","black")
densityplot(~ earconch | sex, groups=Pop,
            data=possum,
            par.settings=simpleTheme(col=colset),
            auto.key=list(space="right"))
```

The functions `densityplot()` and `histogram()` do not allow a name on the left of the ~ symbol. The function `histogram()`, which is otherwise similar to `densityplot()`, does not accept a `groups` argument.

7.2.8 *Panel functions*

Each lattice function that creates a graphics object has its own panel function. Creation of one's own panel function allows detailed control of panel contents. Or `update()` can be used to modify the panel or panels.

Subsection 7.2.9 will describe a radical extension of this basic scheme. Further layers, created using `layer()` and allied functions in the *latticeExtra* package can be "added" (the operator is "+") to a trellis graphics object.

A user panel function will typically include, or consist of, calls to several of the variety of panel functions that are provided in *lattice*. The function `xyplot()` has the panel function `panel.xyplot()`.[12] The following are equivalent:

[12] When a `groups` argument is supplied, `panel.xyplot()` calls the function `panel.superpose()`.

```
xyplot(species ~ length, xlab="", data=cuckoos)
xyplot(species ~ length, xlab="", data=cuckoos,
       panel=panel.xyplot)
```

A user function, used in place of `panel.xyplot()`, might for example call `panel.superpose()`, followed or preceded by other available panel functions.

Available panel functions include:

Note that an alternative to `panel.points()` is `lpoints()`. Similarly for the other functions.

- `panel.points()`, `panel.lines()`, `panel.text()`, `panel.rect()`, `panel.arrows()`, `panel.segments()`, `panel.polygon()` (all documented on the same help page as `panel.points()`);
- `panel.abline()`, `panel.curve()`, `panel.rug()`, `panel.fill()`, `panel.average()`, `panel.mathdensity()`, `panel.refline()`, `panel.loess()`, `panel.lmline()` (all documented on the same help page as `panel.abline()`).

The following graphics object `gph` will be used as a starting point, in the discussion that now follows:

```
gph <- xyplot(Brainwt ~ Bodywt,  data=primates,
              xlim=c(0,300))
```

Now create a panel function that both plots the points and adds labels. The graphics object can then be updated, as in the code that now follows, to use this panel function:

```
my.panel <- function(x,y){
  panel.xyplot(x,y)
  panel.text(x,y, labels=rownames(primates),
             cex=0.65, pos=4)
}
update(gph, panel=my.panel,
       scales=list(tck=0.6))
```

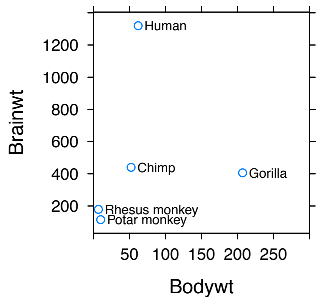


Figure 7.15: Addition of labels, as here, can be done by updating a graph that has the points, by use of a panel function that both plots points and and adds labels, or by adding a new layer.

Note that we could have supplied a panel function that plots the points and adds the labels in the initial function call, thus:

```
xyplot(Brainwt ~ Bodywt, data=primates,
       xlim=c(0,300), panel=my.panel)
```

A further possibility is to add a new layer that has the labels, as in Subsection 7.2.9 which now follows. However the plot is obtained, Figure 7.15 shows the result.

7.2.9 The addition of new layers

The layering mechanism greatly extends the range of possibilities. The code that follows gives a simple and somewhat trivial example of its use – an alternative the use of a panel function for adding labeling to Figure 7.15.

Note again the graphics object gph, created above:

```
gph <- xyplot(Brainwt ~ Bodywt,  data=primates,
              xlim=c(0,300))
```

The following uses the function `layer()`, from the *latticeExtra* package, to create a second layer that has the labels. The layer that is thus created is added to the graphics object gph:

```
gph + latticeExtra::layer(panel.text(x,y,
                          labels=rownames(primates),
                          pos=4))
```

Note also `layer_()`, which reverses the order of the layers, equivalent to using `layer()` with `under=TRUE`.

Other convenience functions are `glayer()` and `glayer_()`. These are equivalent, respectively, to calling `layer()` and `layer_()` with `superpose=TRUE`. The layer is drawn once for each level of any group in the plot.

The function `layer()` allows as arguments, passed via the `...` argument, any sequence of statements that might appear in a panel function. Such statements can refer to panel function arguments, including 'x', 'y' and 'subscripts'. Additionally, named column objects can be passed through an optional `data` argument.

The function `as.layer()` creates a layer from a trellis graphics object. This can then be "added" in the usual way.

7.2.10 Interaction with plots – latticist and playwith

For using *playwith*, the GTK+ toolkit must be installed. For details, go to the website `http://playwith.googlecode.com/`.

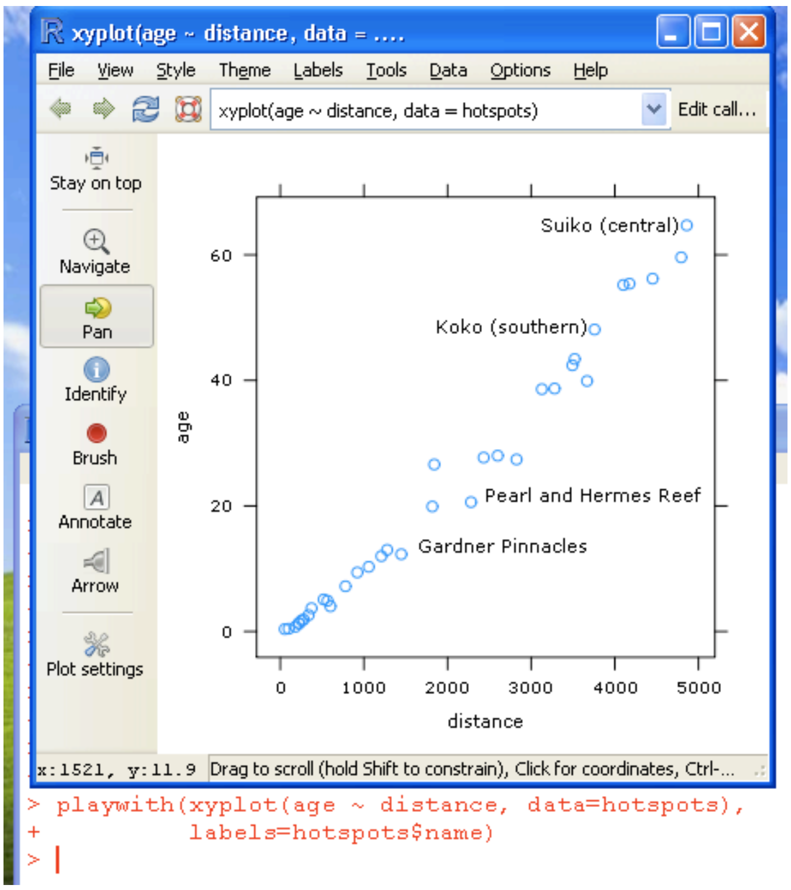
Here will be noted the abilities in the *latticist* and *playwith* packages, for interaction with lattice plots.[13]

[13] More limited abilities are available to interact with other plots.

`latticist()`: When called with a data frame as argument, the function `latticist()` (in the *latticist* package) opens a window that has graphical summary information on the columns of the data frame. Additionally, it opens a GUI interface to the *lattice* and *vcd* packages, allowing rapid creation of plots that may be useful in their own right, or may be a first step in creating more carefully crafted plots. Various annotation features are available from the GUI.

`playwith()`: The `playwith()` function (from the *playwith* package) was used for Figure 7.16. The menu that appears to the left of the graph can be used to initiate single click identification, to add

annotation or arrows, or to mark out a rectangle on the graph for zooming in or out. If labels are not specified, row names are used.


xyplot(age ~ distance, data = ...
File View Style Theme Labels Tools Data Options Help
xyplot(age ~ distance, data = hotspots)
Edit call...
Stay on top
Navigate
Pan
Identify
Brush
Annotate
Arrow
Plot settings
Suiko (central)
Koko (southern)
Pearl and Hermes Reef
Gardner Pinnacles
age
distance
x:1521, y:11.9
Drag to scroll (hold Shift to constrain), Click for coordinates, Ctrl-...

```
> playwith(xyplot(age ~ distance, data=hotspots),
+          labels=hotspots$name)
> |
```

Figure 7.16: This playwith GUI window was generated by wrapping the call to `xyplot()` in the function `playwith()`, then clicking on Identify. Click near to a point to see its label. A second click adds the label to the graph. A color version appears as C.2.

```
## Code that initiates interactive display
library(playwith)
playwith(xyplot(age ~ distance, data=hotspots),
         labels=hotspots$name)
```

Alternative code for Figure 7.16:

```
gph <-
    xyplot(age ~ distance,
           data=hotspots)
library(playwith)
playwith(update(gph),
    labels=hotspots$name)
```

Note that `playwith()` can be used, also, for more limited interaction with plots created using base graphics, or using *ggplot2*.

7.3 *ggplot2 – A Grammar of Graphics*

The *ggplot2* syntax is consistent, but less stylized than the *lattice* syntax. As with *lattice*, *ggplot2* functions return a graphics object. The graphics objects that *ggplot2* functions return can be saved for later use, or updated, or printed directly on to the graphics page. Each different type of *ggplot2* graphic display – scatterplot, histogram, density plot, histogram, etc. – is a different plot `geom`, or "geometry". These can be overlaid.

The *ggplot2* syntax is a variant of Wilkinson's "Grammar of Graphics" (Springer, 2nd edn, 2005).

The following loads the *ggplot2* package:

```
library(ggplot2)
```

7.3.1 *Examples that demonstrate ggplot2 abilities*

Brain weight versus body weight

Figure 7.17 repeats Figure 3.1B from Chapter 3, now using *ggplot2* abilities:

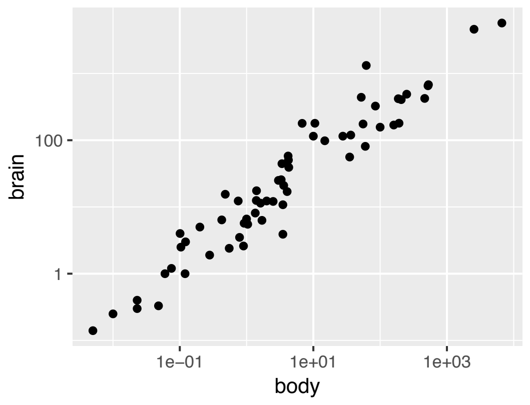


Figure 7.17: Plot of brain weight (gm) versus body weight (kg). Log scales have been used on both axes. The function `coord_equal()`, used with a logarithmic scale, ensures that a given distance (e.g., 1cm) on either axis represents the same relative change.

Code is:

```
library(MASS)
quickplot(body, brain, data=mammals, log="xy") +
  coord_fixed()
```

Notice that `quickplot()` has been used to create an initial plot, with `coord_equal()` then used ("added") to specify that a given distance will represent the same change on both axes. As a logarithmic scale is used, this implies that the same relative change will be given by the same distance. Here, observe that grid lines in both directions are the same distance apart, with the distance representing a change by a factor of 100,

In subsequent discussion, the abbreviated name `qplot` will be used in place of `quickplot`.

The following adds a regression line:

```
quickplot(body, brain, data=mammals, log="xy") +
  coord_fixed() +
  geom_smooth(method=lm)
```

This "addition" of new functions that add to or modify the initial graph can in principle proceed without limit.

As the name hints, the function `qplot()` (or `quickplot()`) shortcuts the more detailed *ggplot2* syntax. The call

```
quickplot((body, brain, data=mammals, log="xy")
```

when written out using the detailed syntactic steps, becomes:

```
ggplot(mammals, aes(body, brain)) +
  geom_point() +
  scale_x_continuous(trans="log") +
  scale_y_continuous(trans="log")
```

The successive "+" operators combine output from function calls to create a single graphics object. In the detailed syntactic steps, the call to `geom_point()` plots the points, while the subsequent calls change the *x*- and *y*-axis scales to logarithmic scales.

In the call to `ggplot()`, the `data` argument is the only mandatory argument. It can be repeated in the call(s) to one or more of the later `geom` functions. This allows different `geom`s, if required, to take their data from different data frames.

Changes to `color` or `size` or `shape` settings can be made separately for each different `geom`. Changing `geom_point()` to `geom_point(size=2.5)` affects only the points.

Note that `cex` and `size` are synonyms, as are `color` and `colour`. Also `type` is a synonym for `geom`.

Aesthetic mappings vs settings

Distinguish between *settings* and *aesthetic mappings*:

	Use of `quickplot()`	Plots based on `ggplot()`
Settings	`size=I(3)` or `cex=3`	`size=3`
Aesthetic mappings	`size=3` or `size=sport`	`aes(size=3)` or `aes(size=sport)`

The function `aes()` maps variables in the data to visual properties ("aesthetics") of `geoms`. In `aes(body, brain` above, the mappings are to the *x*− and *y*−axes of the plot. Other possible mappings are to `color` (use color to distinguish groups within the data), `shape`[14] (distinguish by shape), `size` and `fill`.

[14] Where base graphics has `pch`, *ggplot2* has `shape`.

Use of the argument `size=3` in a call to `quickplot()` does change the point size, but it adds an extraneous key. The same happens if the argument `mapping=aes(size=3)` is supplied to `ggplot()` or to `geom_point()` or to another such function.

A further possibility is to use `quickplot()` (or `qplot()`) to create an initial graphics object, then adding to this object. The following code uses this approach to create Figure 7.20:

```
qplot(Year, mdbRain, data=bomregions2015,
      geom="point",
      xlab="", ylab="Av. rainfall, M-D basin") +
  geom_smooth(span=0.5, se=TRUE)
```

In all cases, a `ggplot` object is created. This can be `print`ed immediately, or it can be saved as a named object. The graph is created using the `print` method for a `ggplot` object.

7.3.2 *An overview of* ggplot2 *technicalities*

Available geometries and settings

Table 7.2 has details of a number of the geometries that are available for `ggplot` objects. Table 7.3 lists some of the settings, in addition to those already noted, that are available:

Table 7.2: Available geoms.

quickplot()	ggplot()	Available arguments to the geom function
geom=		(data, mapping, color, fill, alpha, plus ...)
"point"	+ geom_point()	size, shape, etc.
"line"	+ geom_line()	size, linetype
"path"	+ geom_path()[1]	size, linetype
"smooth"	+ geom_smooth()	linetype, weight, se (TRUE or FALSE).
"histogram"	+ geom_histogram()	linetype, weight
"density"	+ geom_density()	weight, linetype, size
"density2d"	+ geom_density2d()	weight, linetype, size

[1] Use geom_path() to connect observations, in the original order.

Table 7.3: Control of ggplot2 graphics features. Functions such as xlab() and scale_x_continuous() that relate to the *x*-axis all have counterparts with y in place of x. .

	Argument to qplot()	ggplot() or qplot()[1]
Title	main="mytitle"	+ labs(title="mytitle")
Axes	see help(qplot)	+ scale_x_continuous()[2] [or scale_x_discrete() or scale_x_date()]
Axis labels	e.g., xlab="myxlab"	+ xlab("myxlab")[3]
log axes	log="x", (or "y", or "xy")	+ scale_x_log10()[4]
Facets[5]	facets=sex ~ sport	+ facet_grid(sex ~ sport)
Aspect ratio	e.g., asp=1	+ coord_equal()[6]
Theme	—	
Graph title	e.g., main="maintitle"	+ ggtitle("mytitle")

[1] Recall that quickplot() (or qplot()) returns a ggplot object. Functions such as xlab() or scale_x_continuous() can be used, just as for any other ggplot2 object, to update objects returned by quickplot().
[2] Available arguments include limits, breaks (locations for the ticks), labels (labels for the breaks), and trans (e.g., trans="log").
[3] This is an alternative to using name (e.g., name="myxlab") as an argument to scale_x_continuous() or scale_x_discrete().
[4] This is an alternative to using trans="log10" as an argument to scale_x_continuous() or scale_x_discrete(). Note also trans="log" and trans="log2").
[5] Facets give Lattice style *conditioning*.
[6] By default (ratio=1), a given distance, e.g., 1cm, represents the same range along both *x*− and *y*−axes.
[7] Themes control such graphical attributes as background color, gridlines, and size and color of fonts. See help(ggtheme) for details of other available themes.

Example — Measurements on Australian athletes

Figure 7.18 plots height against weight, by sex, for the ais data. Additionally, boxplots show the distributions of heights, and there are two-dimensional density contours estimates. The graph is a tad crowded.

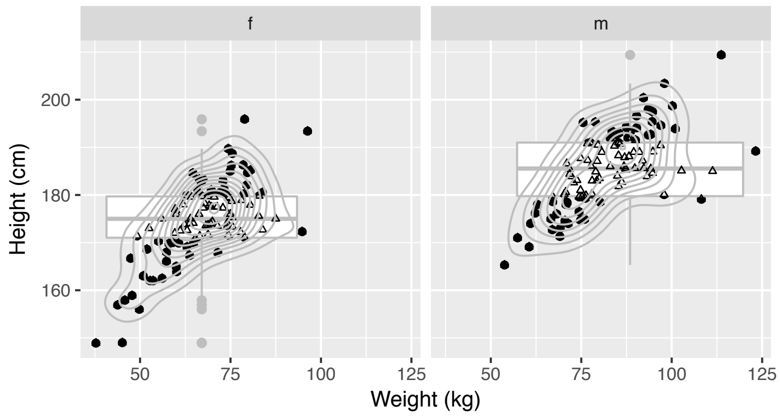


Figure 7.18: Height versus weight, by sex, for Australian athletes in the ais data set. Boxplots that show the distributions of heights, and two-dimensional density contours have been added.

The following gives a simplified version of the plot:

```
## Overlay with boxplots and density contours
quickplot(wt, ht, data=ais,
          geom=c("boxplot", "point", "density2d"),
          facets = . ~ sex)
```

To set axis labels, show the boxplot outline in gray, show contour lines in gray (the default is blue), and make various other changes as in Figure 7.18, specify:

```
quickplot(wt, ht, xlab="Weight (kg)",
          ylab="Height (cm)", data=ais,
          facets = . ~ sex) +
  geom_boxplot(aes(group=sex),
               outlier.size=1.75,
               outlier.colour="gray",
               color="gray") +
  geom_point(shape=2, size=1) +
  geom_density2d(color="gray")
```

The facets argument has the form row.var ~ col.var, where row.var indexes rows of panels, col.var indexes columns, and "." serves as a placeholder when there is one row or one column only.

Code for the next plot will work with a subset of the ais data, limiting attention to rowers and swimmers:

```
## Extract from ais data for rowers and swimmers
aisRS <- subset(ais, sport %in% c("Row","Swim"))
aisRS$sport <- droplevels(aisRS$sport)
```

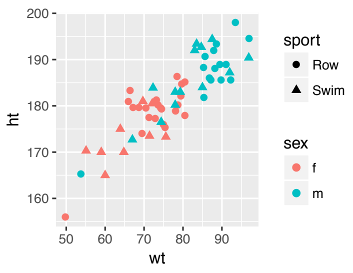


Figure 7.19: Use color for distinguishing sexes, shapes for sports.

Here are alternative code fragments that can be used to create Figure 7.19, one using quickplot() and the other using successive calls to `ggplot()` and to `geom_point()`:

1: Use `quickplot()`:

```
quickplot(wt, ht,
          data=aisRS,
          geom="point",
          size=I(2),
          colour=sex,
          shape=sport)
```

2: `ggplot() + geom_point()`:

```
ggplot(aisRS) +
  geom_point(aes(wt, ht,
               color=sex,
               shape=sport),
             size=2)
```

Multiple aesthetics can be used for the one distinction, here between `sexes`:

```
## Distinguish sex by color & shape
## Different sports have different panels
quickplot(wt, ht, data=aisRS, geom="point",
          size=I(2.5), color=sex, shape=sex,
          facets = . ~ sport)
```

Here are further possibilities:

```
## Identify sex by color, sport by shape (1 panel)
quickplot(wt, ht, data=aisRS, geom="point",
           color=sex, shape=sport, size=I(2.5))
## Identify sex by color, sport by size (1 panel)
quickplot(wt, ht, data=aisRS, geom="point",
          color=sex, size=sport)
```

Australian rain data

Figure 7.20 plots annual rainfall for Australia's Murray-Darling basin region. The following code uses the function `quickplot()`:

```
library(DAAG)
library(ggplot2)
## Default loess smooth, with SE bands added.
quickplot(Year, mdbRain, data=bomregions2015,
          geom=c("point","smooth"), xlab="",
          ylab="Av. rainfall, M-D basin")
```

Arguments `size` (e.g., `size=I(2.5)`), `color` (e.g., `color=I("red")`, etc, can be supplied, affecting both points and the added smooth curve. NB: `size=I(2.5)`, not `size=2.5`.

Code that shows the detailed syntactic steps is:

```
ggplot(bomregions2015, aes(x=Year, y=mdbRain)) +
  geom_point() +                    # Scatterplot
  geom_smooth(span=0.5, se=TRUE) +  # Add smooth
  xlab("") +             # Blank out x-axis label
  ylab("Av. rainfall, M-D basin")
## NB: aes() has supplied x- and y-axis variables
```

As before, the successive "+" operators combine output from function calls to create a single graphics object.

Try also the following. This requires both the *quantreg* package and the *splines* package:

```
library(quantreg)
library(splines)
## Supplementary figure 4
quickplot(Year, mdbRain, data=bomregions2015) +
```

The normal spline basis `ns(x,5)` is supplied to the function that estimates the quantile curves, so that 5 d.f. spline curves are fitted at the 20%, 50% and 80% quantiles.

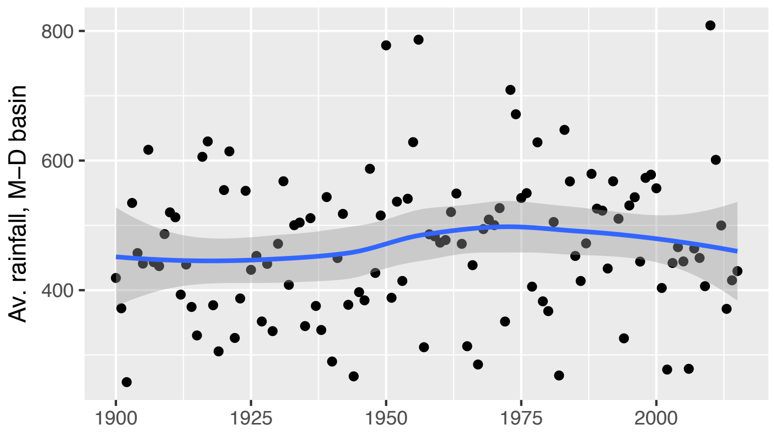


Figure 7.20: Annual rainfall, from 1901 to 2012, for the Murray-Darling basin region of Australia. The curve is fitted using the default loess smoother. The pointwise standard error bands assume that errors about the curve are independent; this is unlikely to be strictly true. To suppress these bands, specify `se=FALSE`.

```
          geom_quantile(formula = y ~ ns(x,5),
          quantiles=c(0.2,0.5,0.8) )
```

Florence Nightingale's Wedge Plot

Figure C.3 in Appendix C is a "wedge" plot, showing the mortality of British troops according to cause in the Crimean war over 1854–1856. It shows the abilities of the *ggplot2* package to spectacular effect. The plot is obtained by using polar coordinates for plotting a stacked bar chart! Use of areas to convey numerical information is however not ideal, especially when as here the areas overlap.

Florence Nightingale's Crimean War experience prepared her for later major work in the reform of army and civilian hospitals and public health administration, and to wider social reform. Her influence extended to the army and civilian administration in India.

7.4 Static graphics – additional notes

7.4.1 Multiple graphs on a single graphics page

For *base* graphics, refer back to Subsection 7.1.5. The following demonstrates use of the `fig` argument to `par()` to select a part of the display region for plotting:

```
par(fig = c(0, 1, 0.38, 1))
          # xleft, xright, ylow, yhigh
## Plot graph A
par(fig = c(0, 1, 0, 0.38), new=TRUE)
## Plot graph B
par(fig = c(0, 1, 0, 1))      # Resets to default
```

For lattice graphs, the location of the graph can be determined by the argument `position`, when `print()` is called. The following demonstrates its use:

```
cuckoos.strip <- stripplot(species ~ length, xlab="", data=cuckoos)
print(cuckoos.strip, position=c(0,0.5,1,1))
                   # xleft, ybottom, xright, ytop
cuckoos.bw <- bwplot(species ~ length, xlab="", data=cuckoos)
print(cuckoos.bw, position=c(0,0,1,0.5), newpage=FALSE)
```

Note the use of `newpage=FALSE` for the second plot.

Base and trellis plots on the same graphics page

The following uses the base graphics command `mtext()` to label a lattice plot:

```
plot(0:1, 0:1, type="n", bty="n", axes=FALSE,
     xlab="", ylab="")
lab <- "Lattice bwplot (i.e., boxplot)"
mtext(side=3, line=3, lab)
cuckoos.bw <- bwplot(species~length, data=cuckoos)
print(cuckoos.bw, newpage=FALSE)
```

Inclusion of graphs in Microsoft Word

Graphs may not import well from the clipboard into Word on the Macintosh under OS X. On Windows systems, an effective option is to use `win.metafile()` to write graphics output to a Windows metafile format that should import without problem into a Word or Power Point document.

7.5 Dynamic Graphics – rgl

This section will describe a range of abilities that create displays which the user can then manipulate dynamically. Note in particular the *rgl* and *googleVis* packages, designed for interactive exploration of dynamic changes in relationships with time. The *googleVis* package reproduces most of the abilities of Google's Public Data Explorer, which can be accessed at `http://www.google.com/publicdata/home`

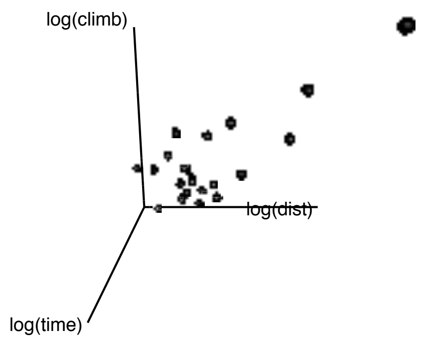


Figure 7.21: Snapshot of a 3-D dynamic display, for the `nihills` data from the *DAAG* package. The display has been dragged to a position where points very nearly fall on a line.

The *rgl* package provides three-dimensional dynamic graphics. Use of the functions `scatter3d()` and `identify3d()` from the *car*

package may be more convenient for novices than the *rgl* functions that they call. Figure 7.21 shows a snapshot of the plot obtained for the `nihills` data from the *DAAG* package.
The following loads the needed packages:

```
## The car and rgl packages must be installed
library(rgl, quietly=TRUE)
library(car, quietly=TRUE)
rgl::setupKnitr()
knit_hooks$set(rgl=hook_rgl)
```

Code for the figure is:

```
library(DAAG, quietly=TRUE)
open3d()             # Precedes the call to par3d()
par3d(cex=0.75)      # Optional
                     # Other params: see help(par3d)
with(nihills, scatter3d(x=log(dist), y=log(climb),
                        z=log(time),
                        grid=FALSE,
                        surface=FALSE,
                        point.col="black",
                        axis.scales=FALSE))
## NB: Use middle or right mouse button to drag a
## rectangle around a point that is to be labeled.
```

Use the function `identify3d()` to start the identification of points. Following the call to `identify3d()`, use the middle (or maybe right) mouse button to drag a rectangle around any point that is to be labeled. To cease identifying points, make a middle (or right) click on an empty region of the plot. The labels may appear only at this point. Here, for identification of points shown in Figure 7.21, is suitable code:

Use `rgl.snapshot()` to save the current plot into a file.

```
with(nihills, identify3d(x=log(dist), y=log(climb),
                         z=log(time),
                         labels=row.names(nihills),
                         col="gray"))
```

Such a plot can be helpful in identifying high leverage points, e.g., in the regression of `log(time)` on `log(dist)` and `log(climb)`. The plot needs to be rotated to give a view in which the leverage is apparent.

The `rggobi` *package*

The *rggobi* package offers a wider range of features, via an interface to the GGobi system. For installation details go to `http://www.ggobi.org/`. Windows users can use the following to install all the required files from an R session that has access to a live internet connection:

```
source("http://www.ggobi.org/downloads/install.r")
```

7.5.1 *The* googleVis *Package*

This provides an interface to the abilities of Google's Public Data Explorer. While these abilities can be accessed from the web page noted below, there are obvious advantages in setting up the display from one's own computer.[15]

[15] An internet connection is needed to access Google's API (Application Program Interface) when the chart is displayed.

Google's Public Data Explorer

Upon accessing Google's web page http://www.google.com/publicdata/home, the display will cycle through examples of the use of Motion Charts, and other related charts. Click on Explore Data to go to the interactive version of the relevant display.[16] These charts, which are interesting in themselves, show the abilities that *googleVis* is designed to emulate. See the annotations in Figure C.1 below for details that should be enough to get started.

[16] Various controls are placed in the margins of the graph. Move the pointer over one or other control feature to get information on its purpose, or over a point to display information about that point. For changing the *x*- and/or *y*- variables, or for changing the variable that determines point size, click on the relevant downward pointing selector arrow. Scales, separately for the two axes, can be either linear or logarithmic.

A slider below the graph can be moved to show how the relationships that are plotted change over the available timespan, commonly 1960 through to 2010 (but note that not all data will be available for all years). Click on the solid right-pointing triangle on the left of the slider scale to see the graph changing dynamically in moving from the currently shown year through to 2010.

Use of googleVis *to create motion charts*

The details given here should be supplemented with examination of the vignette that accompanies the *googleVis* package. Note especially Figure 1 on page 5 of the vignette.

To display the vignette, type: `vignette("googleVis")`

Creation of a motion chart, once the data is in place, is remarkably straightforward. The starting point is a data frame in that has a column (e.g. Countries) that can be used as an `id` variable, a column (e.g. Year) that has a `timevar` variable, and columns that can be used to supply *x*- and *y*- variables. Optionally, columns may be identified for use as a `colorvar` and/or a `sizevar`.

The dataset `grog` from the *DAAG* package has a suitable structure. One can create a motion chart thus:

```r
library(googleVis)
M <- gvisMotionChart(grog, id="Country", timevar="Year")
## This next line requires a live internet connection,
## and Adobe Flash must be installed.
plot(M)
```

If the browser window that appears displays 'Flash' in gray in the middle of the screen, click there to proceed. A browser window with a gray display region should appear.

For the `grog` dataset, the Motion Chart does a less satisfactory job than Figure 7.10 in Section 7.2.4. Motion charts come into their own for the examination of steady changes over time in a bivariate relationship, with different patterns of relationship for different subgroups of the data.

A plot that allows the display of various World Bank development indicators can be obtained by typing:

```
demo(WorldBank)
```

The code used to download the data and display the motion chart will appear on your screen.

This can take a while to start up – data has to be downloaded from the World Bank web site. Hover the mouse pointer over features that appear in the margins of the display to see annotation that indicates how you can change or manipulate various aspects of the display.

The data from the World Bank site is stored into a data frame `WorldBank`. The command that creates a gvis object M is:[17]

[17] Both `WorldBank` and `M` should be in your workspace after running `demo(WorldBank)`. The data are also alternatively available from the image file `WorldBank.RData` at the url noted on the reverse of the title page.

```
M <- gvisMotionChart(WorldBank, idvar="country",
          timevar="year",
          xvar="life.expectancy",
          yvar="fertility.rate",
          colorvar="region", sizevar="population",
          options=list(width=700, height=600))
## Now display the motion chart
plot(M)
```

Change `width` and `height` as needed to make better use of the screen display.

If arguments are supplied, security setting issues on the user computer can result in an initial assignment of columns that does not accord with the supplied arguments.[18] The drop-down menus should however function correctly, and can be used to obtain a display that accords with any choice of arguments that the user may want.

[18] The *gvis* object M comprises Javascript code that can be included on a web page. This should display correctly when the web page is accessed.

For a further example, load the image file **wdiSel.RData**, available from the url noted on the reverse of the title page. This will make available the data frame `wdiSel`. This has a larger number of indicators, but for 26 countries only. Figure C.1 (with the figures that are shown in color) shows an annotated version of a motion chart that was created from this dataset.

The following code generated the initial chart. The change to a log scale on the vertical axis was made interactively:

```
xnam <- "Electric power consumption (kWh per capita)"
ynam <- "Mobile cellular subscriptions (per 100 people)"
M <- gvisMotionChart(wdiSel, idvar="Country.Name", timevar="Year",
                     xvar=xnam, yvar=ynam,
                     colorvar="region", sizevar="Population, total",
                     options=list(width=600, height=500),
                     chartid="wbMotionChartSel")
plot(M)
```

7.6 Summary

Base graphics functions plot a graph. Lattice and ggplot2 functions return a graphics object. which can then stored or updated or plotted (printed).

A powerful feature, both of *ggplot2* graphics and of *lattice* graphics when the layering abilities of the *latticeExtra* package is used, is the ability to build a graph up layer by layer.

The R system makes available, via its various package, a wide variety of other graphics abilities. This includes dynamic and other 3-dimensional graphics.

7.7 Exercises

In the following exercises, if there is no indication of whether to use `base` or `lattice` graphics, use whichever seems most suitable.

1. Exercise 3 in Section 2.6.2 showed how to create the data frame `molclock`. Plot `AvRate` against `Myr`. Use `abbreviate()` to create abbreviated versions of the row names, and use these to label the points.

2. Compare the following graphs that show the distribution of head lengths (`hdlngth`) in the `possum` data set. What are the advantages and disadvantages of these different forms of display?

   a) a histogram (`hist(possum$hdlngth)`);

   b) a stem and leaf plot (`stem(qqnorm(possum$hdlngth))`;

   c) a normal probability plot (`qqnorm(possum$hdlngth)`); and

   d) a density plot (`plot(density(possum$hdlngth))`.

3. This exercise uses the data set `hotspots` (*DAAG* package).

   Plot `age` against `distance`. Use `identify()` to determine which years correspond to the two highest mean levels. That is, type

   ```
   plot(age ~ distance, data=hotspots)
   with(hotspots, identify(age ~ distance, labels=name))
   ```

   Use the left mouse button to click on the highest two points on the plot. (Right click in the figure region to terminate labeling.)

4. Use `mfrow()` to set up the layout for a 3 by 4 array of plots. In the top 4 rows, show normal probability plots for four separate 'random' samples of size 10, all from a normal distribution. In the middle 4 rows, display plots for samples of size 100. In the bottom four rows, display plots for samples of size 1000. Comment on how the appearance of the plots changes as the sample size changes.

5. The function `runif()` can be used to generate a sample from a uniform distribution, by default on the interval 0 to 1. Print out the numbers you get from `x <- runif(10)`. Then repeat exercise 6 above, but taking samples from a uniform distribution rather than from a normal distribution. What shape do the points follow?

6. The data frame `airquality` that is in the base package has columns `Ozone`, `Solar.R`, `Wind`, `Temp`, `Month` and `Day`. Plot `Ozone` against `Solar.R` for each of three temperature ranges, and each of three wind ranges.

7. Create a version of the data frame `Pima.tr2` that has `anymiss` as an additional column:

```
missIND <- complete.cases(Pima.tr2)
Pima.tr2$anymiss <- c("miss","nomiss")[missIND+1]
```

(a) Use strip plots to compare values of the various measures for the levels of `anymiss`, for each of the levels of `type`. Are there any columns where the distribution of differences seems shifted for the rows that have one or more missing values, relative to rows where there are no missing values?

Hint: The following indicates how this might be done efficiently:

```
library(lattice)
stripplot(anymiss ~ npreg + glu | type, data=Pima.tr2, outer=TRUE,
          scales=list(relation="free"), xlab="Measure")
```

(b) Density plots are in general better than strip plots for comparing the distributions. Try the following, first with the variable `npreg` as shown, and then with each of the other columns except `type`. Note that for `skin`, the comparison makes sense only for `type=="No"`. Why?

```
## Exercise 7b
library(lattice)
## npreg & glu side by side (add other variables, as convenient)
densityplot( ~ npreg + glu | type, groups=anymiss, data=Pima.tr2,
            auto.key=list(columns=2), scales=list(relation="free"))
```