Base graphics in R

Antonio Gasparrini

01 November 2023

Contents

The R software is well-known for its excellent graphical facilities. Here, we provide a brief summary of the main features of the base graphics tools implemented in the standard R packages. In this framework, the emphasis is in the provision of simple and short expressions to quickly obtain unrefined plots, while more sophisticated and refined versions are obtained through a vast assortment of graphical parameters, all of which can be modified by the user. The result is an approach to graphics which encourage the exploration of different graphical tools, more than offering a set of inflexible methods. The first section gives an overview of the graphical facilities in R, while the second one introduces the main plotting function. Then, the use of graphical parameters and other plotting tools are described, with a distinction between high and low-level functions. The last section illustrates how graphs can be exported outside R.

Overview of base graphics in R

The R software offers a vast collection of facilities for creating graphics, from simple, intuitive scatter plots to complex three-dimensional figures. Similarly to other computations, also graphs are produced and managed in R through calls to specific functions, available in different packages. This array of tools is built upon two main graphic systems within R: the *traditional* system and the *grid* system. Both are based on a graphic engine and devices included in the standard package GRDEVICES, and both offer functions to create a wide selection of types of graphics. Their computational scheme is rather different, and in particular the grid system provides a more evolved framework. However, from a purely practical perspective, the range and quality of graphics you can create with the two systems are rather similar. In this session, we focus on the traditional (or *base*) system implemented through the standard package GRAPHICS. Implementations of the grid system, for example through the recommended packages GRID and LATTICE, are not described here.

The R functions producing graphs within the traditional system are somewhat different from the other functions. When called, they do not return a result which can be saved in a new R object through assignment. This is not required, as the interest lies in their side effect, which is to produce a graphical element in a device. In particular, the main functions usually open a new screen device where the graph is displayed. In the standard R GUI, the screen device is opened as an additional sub-window, while in RStudio the device is shown in the PLOTS window.

Methods for building base graphics in R follow an *ink-on-paper* model: graphical elements are added progressively to a graphic device, by consecutive calls to different functions. Specifically, two main types of plotting functions exist in R. *High-level* functions plotting function} by default open a new plot in the device (therefore erasing everything plotted earlier) and produce a complete plot. *Low-level* functions instead add graphic elements to an existing plot produced by a high-level function. Therefore, initial versions of graphs are planned on purpose to be elementary and suitable for further additions.

This ability is a fundamental feature of base R graphics, and it is consistent with the computational flexibility of the software. You will find out how building complex, refined, publication-level graphs in R is an easy task, although it may require a relatively long code and the use of several graphic functions and arguments. Coherently with the R philosophy, graphical facilities are not built for performing standard, fixed actions, but offer a general and flexible framework where the user can experiment and create his or her personal versions of graphs.

The plot function

The main function for producing graphs in R is `plot()`. Its basic use is for creating *scatter plots*, namely diagrams of Cartesian coordinates to display the relationship between two variables. This function allows different syntaxes. The most straightforward one includes two main arguments `x` and `y`, identifying numeric vectors of the same length corresponding to the *(x,y)* coordinates, namely the values in the *x* and *y*-axis, respectively. A second syntax includes a formula defining an expression like `y ~ x` with the *tilde* symbol $\sim$. A second optional argument `data` specifies a data frame where these numeric vectors can be stored.

As an example, we use `plot()` to obtain a scatter plot using data in the data frame BIRTHS in the package EPI. We first load the package (which might also be installed, if not):

```
library(Epi)
data(births)
```

Specifically, we are interested in assessing the relationship between birth weight and gestational week. We create the plot using an expression involving the syntax with `formula` and `data`:

```
plot(bweight~gestwks, data=births)
```

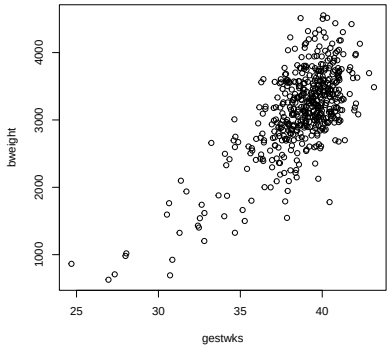

Figure 1: A basic scatter plot created with the function plot()

The alternative usage and syntax would rely on `x` and `y` arguments, for instance `plot(births$gestwks, births$bweight)`. Note that the function `plot()` does not require the two numeric vectors to be stored in a data frame, and also works with objects independently created in the global environment. In this case, the argument `data` would not be needed.

The resulting graph appears as a new page in the PLOT windows of RStudio. As anticipated earlier, the graph is elementary, with points identifying the plot coordinates corresponding to each observation of `gestwks`. These points are placed within a frame, enclosed by axes with ticks, annotation and labels. The graph is in black and white, with no colours. However, this simple plot is enough to perform a first assessment of the relationship between the two variables.

Each graphic page in the traditional graphic system has three main regions: the *outer margins*, the *figure region* and the *plot region*. The outer margins are usually zero by default, while the region obtained by excluding them is called *inner region*. When only one figure is displayed, as in the previous example, the inner region corresponds to the figure region. However, when multiple figures are displayed, the inner region is the union of multiple figure regions. A figure region includes the figure margins, where the axes and labels are displayed, and the plot region, which in the previous example corresponds to the area within the axes.

The `plot()` function has an object-oriented behaviour, meaning that it produces different plots depending on the class of the objects used as arguments (which can also be the left or right-hand side of the formula). For example, let's try to use it again with the dataset AIRQUALITY in the package DATASET, plotting the series of temperatures in New York in June 1973 (see `help(airquality)`). We call `plot()` using the formula syntax, including the proper subset of records in `data`:

```
plot(Temp ~ Day, data=subset(airquality, Month==6))
```

As another example, we apply `plot()` with the dataset MTCARS in the same package, this time calling it with a single argument represented by the variable `carb`, which reports the number of carburettors in a car. However, this time we transform it into a categorical variable using `factor()`:

```
plot(factor(mtcars$carb))
```

Finally, we use it again with the dataset ESOPH, examining the association between the number of cases stored in the variable `ncases` and the age groups defined by `agegp` (see `help(esoph)`). The code is:

```
plot(ncases ~ agegp, data=esoph)
```

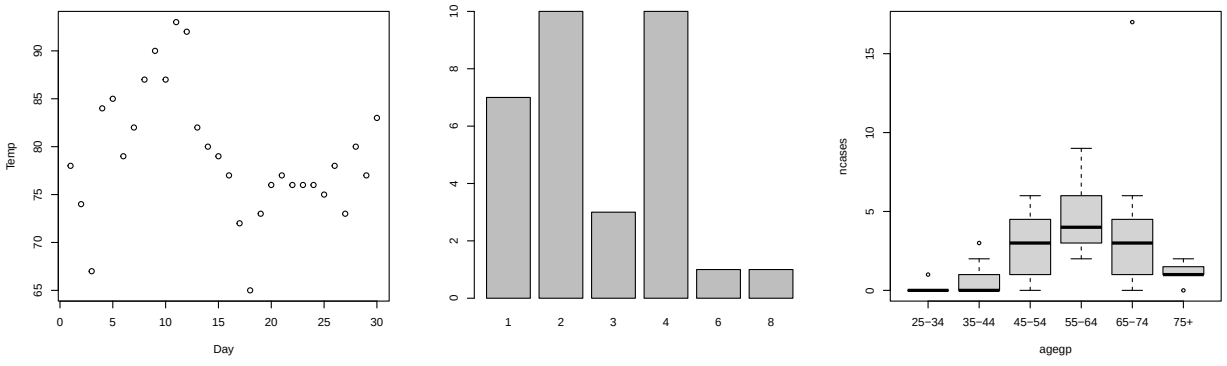

Figure 2: A simple scatter plot (left), a bar plot (mid), a box plot (right)

The result of the three calls produces completely different graphs. In the first example, `plot()` is called using two numeric vectors, and therefore it produces a simple scatter plot (left panel) where the value of each element of `Temp` (in the *y*-axis) is plotted versus `Day` (in the *x*-axis). In contrast, in the second example, `plot()` is called with a factor, and it creates instead a *bar plot* (mid panel) identifying the occurrences of

each category in `carb`. Finally, the third call features a numeric variable and a factor, and in this case `plot()` produces a *box plot* (right panel), also known as *box-and-whiskers* plot, which is the straightforward method to display the distribution of the quantitative variable `ncases` within the multiple categories of `agegp`.

This flexibility is provided by the fact that `plot()` is defined as a *generic* function that calls internally other *method* functions specific to the class of the object(s) used in its first argument. With this approach, the graph obtained is relevant for summarizing the specific information stored in the object, using a simple and consistent syntax to obtain various graphical outputs.

RStudio simplifies substantially the management of graphs in R. For instance, you may have already noted that at any new call of a function, a new graph is displayed in the PLOTS window. The 'arrows' buttons allow retrieving previous graphs and navigating back and forward among the pages. The button ZOOM can be used to open an external page with the same graph, while the last two buttons erase the last plot or clear the window.

Graphical parameters

The graphs created in the previous section are clear enough to be interpreted, and even in this elementary format, they already provide information on some characteristics of the data we are analysing. However, some features of these plots are scanty and inadequate to communicate the results effectively.

For example, even in these basic graphs, the labels of the axes may appear obscure, even when provided, and the range of the axes is sometimes not optimal. Also, the appearance is unrefined, and these graphs must be improved before being presented, for instance, in published papers or books.

Luckily, the graphs in R are specified by a vast assortment of *graphical parameters* which can be adapted to modify the look of the plots. In particular, graphical parameters can be changed from their default values in two ways: through arguments of the plotting functions or through the `par()` function. Here you can find examples with a (non-exhaustive) illustration of the use of graphical parameters.

We can start with the former method, using arguments in `plot()` to change specific graphical parameters, with the aim to restyle some of the graphs previously created. For instance, see the code:

```
plot(bweight~gestwks, data=births, xlab="Gestational week", ylab="Birth weight (gr)",
  xlim=c(25,45), pch=19, frame.plot=FALSE, col="red", cex=0.7)
```

This creates the same graph seen before, but with some non-default graphical parameters. Specifically, the arguments `xlab` and `ylab` accept strings, used to create the labels for the axes. The argument `xlim` accepts a numeric vector of length 2 setting the range of the *x*-axis (another argument `ylim` does the same for the *y*-axis). The argument `pch=19` states that the points must be plotted as solid circles, while `frame.plot=FALSE` excludes the default box around the plot region. The red colour and size (relative to 1) of the plotting elements are set through `col` and `cex`. See `help(plot.default)` and `help(points)` (described later) for additional details and a complete list of the available arguments.

Another example illustrates changes in the other scatter plot created in the previous section:

```
plot(Temp ~ Day, data=subset(airquality, Month==6), type="o", ylab="Temperature",
  ylim=c(55,105), lty=2, lwd=1.5, pch=22, bg="green", cex=1.4)
```

Here the argument `type="o"` (default to `"p"`, as points) specifies that lines connecting points must be drawn (another common choice is `"l"` for lines). The type and width of the lines are set by `lty` and `lwd`, respectively. The value 2 for the former specifies dotted lines, while the latter is relative to 1. This time, points are drawn as squares (`pch=22`) with a background colour defined by `bg`. See `?plot.default` and `?lines` (described later) for additional information.

In a final example, we modify the box plot created in the previous section. The code:

```
plot(ncases~agegp, data=esoph, xlab="Age group", ylab="Number of cases", ylim=c(0,10),
  col=4, cex.axis=1.2, cex.lab=1.5, main="Number of cases by age categories")
```

In this case, also the size of axis annotations and labels has been extended (relative to 1) with the arguments `cex.axis` and `cex.lab`, respectively. The argument `main` adds a title to the top of the plot.

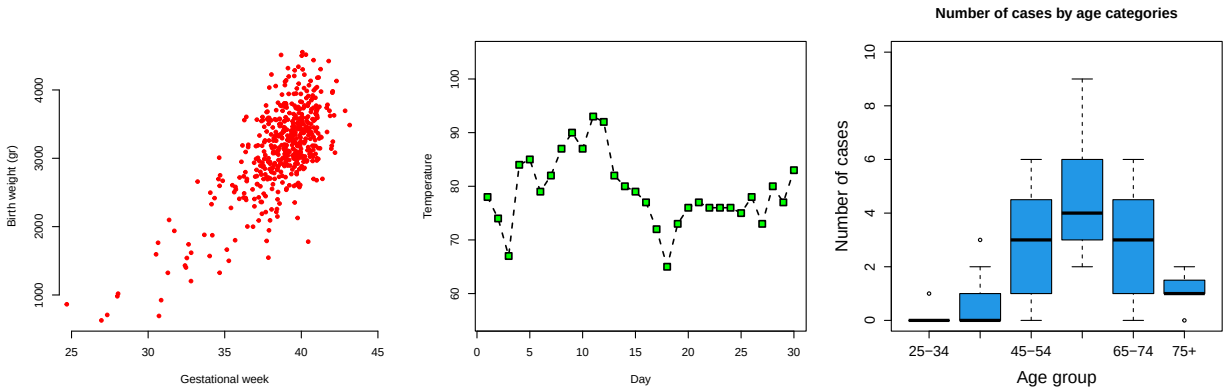

Figure 3: A simple scatter plot (left), a bar plot (mid), a box plot (right)

Also, note how in this last example the colour for the plotting elements (in this case the blue for boxes of the box-and-whisker plot) has been set with a number, instead of with strings as in the previous plots. The two methods are interchangeable and the number is converted to a colour using the function `palette()`, which provides eight colours which can be called quickly with numbers (see the related help page).

The traditional graphic system of R comes with a pre-defined set of graphical parameters, defining a *graphic state* which is internally consulted to determine the features of a graph. The `par()` function is used to access, display and/or modify such a state, *i.e.* by modifying the default values of graphical parameters. Some of these parameters are the same included as arguments of the `plot()` function seen above, but others are different and can only be inspected or set through `par()`. For instance:

```
par(c("cex","col"))
```

```
## $cex
## [1] 1
##
## $col
## [1] "black"
```

The code returns the default values for two of the parameters seen earlier. In order to change these default values, parameters can be set by specifying them as tagged arguments of `par()`, for instance `par(cex=1.5, col="blue")`. Type `par()` for the complete list of graphical parameters (with related default values), and `help(par)` for more details. Be aware that, differently from the arguments of the plotting functions, changes provided by `par()` are permanent, unless the graphic device is closed (for example with the CLEAR ALL button in the Plots window of RStudio)

Other high-level functions

Although flexible, `plot()` is not the only high-level plotting function available in R. Many more exist in the package GRAPHICS and in other contributed packages. We illustrate here some of the most common ones. This will also offer the opportunity to describe other graphical parameters, either specific to each type of plotting functions or more general.

As a first example, we show how to examine the distribution of a continuous variable using `hist()` and `qqnorm()`. Specifically, we evaluate them with the variable `bweight` in the data frame `births` introduced earlier. The code:

```
hist(births$bweight, breaks=20, col=2, xlab="Birth weight (gr)", main="An histogram")
qqnorm(births$bweight)
```

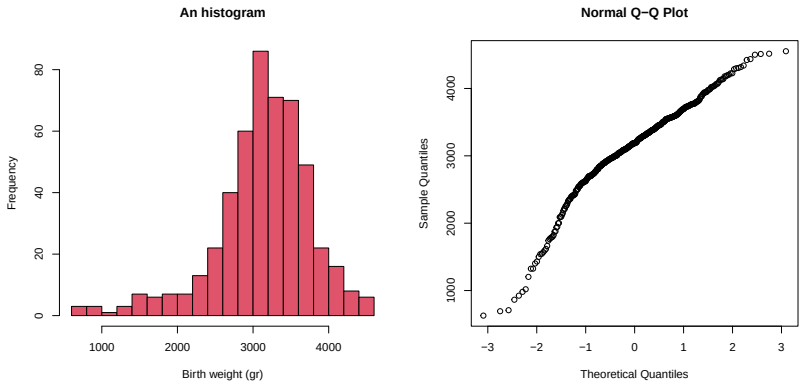

Figure 4: A bar plot (left) and normal QQ plot (right)

The function `hist()` produces a *histogram* of the values included in the vector inputted as its first argument, using an approximate number of bins specified in the other argument `breaks`. This function has many other arguments: see `?hist` for more details. The second graph created by `qqnorm()` is a *normal Q-Q plot*, useful to check the distribution of the variable versus the theoretical percentiles of a reference distribution, in this case the Gaussian distribution.

Other high-level functions are used to visualise distributions of categorical variables, for instance `barplot()` and `pie()`. The former draws bar plots, extending the capability of function `plot()` illustrated in previous sections for this type of graph. This function accepts as its first argument either a numeric vector or a matrix, or objects which can be coerced as such, representing the frequencies of categorical variables. Let's see an example with the variable `gear` in the data frame `mtcars`, whose frequencies can be computed directly using the function `table()`:

```
barplot(table(mtcars$gear), horiz=TRUE, las=1, xlab="Frequency",
  ylab="Number of gears", main="A bar plot")
```

In this expression, specific arguments of `barplot()` can be used, such as `horiz` (default to `FALSE`) which draws a horizontal plot. The general argument `las` can be used to force the appearance of the axis annotation, in this case drawn as horizontal (see `help(par)`). The function `barplot()` can be applied to create more complex bar plots. Check `help(barplot)` for additional information.

Alternatively, the function `pie()` can be called to create *pie charts*, where the frequencies are translated into relative areas of each slice of the pie. Let's repeat the previous example:

```
pie(table(mtcars$gear), col=2:4, main="A pie chart")
```

As a final example, we illustrate the use of the function `boxplot()`, which has extended the capabilities to draw this type of graph. The function has many usages, and here we apply it with a formula, creating a box plot similar to that produced in a previous section with `plot()`. The code:

```
boxplot(ncases~agegp, data=esoph, xlab="Age group", ylab="Number of cases",
  ylim=c(0,10), col=4, boxwex=0.5, main="A box plot")
```

Here, the argument `boxwex`, specific to `boxplot()`, is used here to reduce the dimensions of the boxes. See `help(boxplot)` for the complete list of arguments and more information.

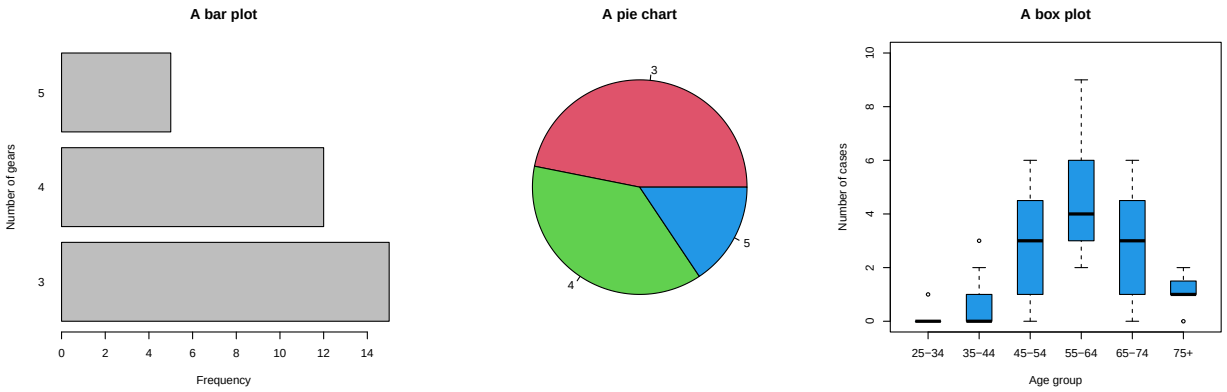

Figure 5: A bar plot (left), normal QQ plot (mid), and a box plot (right)

Many other high-level functions are available in R, most of them in contributed packages. In particular, other methods for the generic function `plot()` are developed for new classes of objects, providing a wide range of types of graphs. Explore the help pages of functions and packages for additional details.

Low-level functions

The use of arguments in high-level functions and in `par()` allows fine-tuning of the graphs created with R. However, this approach to graphing is still somewhat inflexible, as it requires to specify all the features in a single call to a high-level function. Sometimes, it is easier and more efficient to start with a very basic version of the graph, and then add graphical elements with the use of low-level plotting functions.

Similarly to their high-level counterpart, many low-level functions are available in R. We will illustrate here the use of the most common ones, suggesting beforehand to have a look at the help page of each function for more details. The best way to make the most of this approach to graphing is to start with an almost blank graphic page, and then to call different functions to add single elements.

The first example uses the same scatter plot we produced in a previous section. In this case, we want to repeat the graph plotting the birth weight versus gestation calculated in months, instead than weeks using the conversion ratio 4.33 weeks in a month. We anticipate that this example is silly, as this conversion can be done beforehand creating a new vector with gestational month. However, the example is useful to illustrate the concept. We start with:

```
plot(bweight~gestwks, data=births, type="n", xlab="", ylab="Birth weight (gr)",
  xlim=c(25,45), frame.plot=FALSE, xaxt="n")
title(main="Birth weight vs. gestational month", xlab="Month")
axis(1, at=6:10*4.33, labels=6:10)
points(bweight~gestwks, data=births, pch=23, bg=7)
```

This first call to the high-level function `plot()` produces an almost empty graph (left panel) by using special arguments to eliminate some of the graphical elements. Specifically, `type="n"` excludes the main graphical elements, in this case the points. The label for the *x*-axis is excluded instead by passing an empty string to `xlab`. The argument `ann` can instead be used to eliminate the labels for both axes. The consequence of `frame.plot=FALSE` has already been commented, while the effect of `xaxt="n"` is to avoid the plotting of the *x*-axis itself. The argument `yaxt` does the same for the *y*-axis, while setting `axes=FALSE` excludes both axes.

The second and third expressions are calls to low-level functions that add the titles/labels and the *x*-axis to the existing graph (mid panel). The function `title()` has arguments `main`, `xlab` and `ylab`, amongst others,

Figure 6: Building a graph step by step with low-level functions

to set the title and axis labels. Note how these are the same arguments used for the same purposes in `plot()` as well. The function `axis()` has a long sequence of arguments to finely tweak the look of the axis. Here we use `at` to select where the tick marks should be annotated, and `lab` for their labels. The sequence of months from 6 to 10 is conveniently converted in weeks for being plotted in the underlying grid.

The last step is to add the main graphical elements with the low-level function `points()` (left panel). This low-level function has a usage very similar to `plot()`, with almost the same arguments. In particular, here we choose an alternative pair for the `pch` and `bg` arguments, selecting the type and background colour for the points. Similarly, the function `lines()` can be used to add lines instead of points.

Another useful low-level function is `abline()`, which adds one or more straight lines through the current plot. We try it out with the normal Q-Q plo previously created:

```
qqnorm(births$bweight)
abline(v=0, lty=5)
abline(h=median(births$bweight), lty=5)
```

The function has different usages depending on the arguments used (see `help(abline)` for more examples. The calls above draw horizontal and vertical lines at the specific coordinates corresponding to arguments `h` and `v`, which are set to 0 and the median of `bweight`, respectively (left panel in the figure below).

In the next example, we use low-level functions which add other graphical elements, starting again from an almost empty page:

```
plot(0:10,0:10, type="n", ann=FALSE)
legend("topright", c("a line","a point"), lty=c(2,NA), pch=c(NA,19),
  col=2:3, inset=0.05)
segments(x0=1:4, y0=6.5:9.5, x1=6, y1=6.5:9.5, col=4)
polygon(x=1+0:40/10, y=3+(-20:20/15)^3, col=2)
text(x=8, y=5:2, labels=c("We draw:", "- a legend", "- some segments", "- a polygon"))
```

Specifically, the function `legend()` includes a legend in the plot (right panel), while `segments()` and `polygon()` draw segment lines and polygons, respectively. Finally, `text()` can be used to add some text. Interestingly, the last three functions are vectorized and accepts vectors as arguments for the coordinates for plotting multiple elements or complex shapes like polygons. Refer to the help pages for information on the usage and specific arguments.

There are alternative methods to add graphical elements. One is to treat high-level functions as low-level ones. This can be done when the argument `add` is available, or alternatively by setting the graphical parameter `new` to `TRUE` through `par()`. This prevents the old plot from being erased in the graphic device as soon as the the new high-level function is called. We can use the former method to modify the histogram created

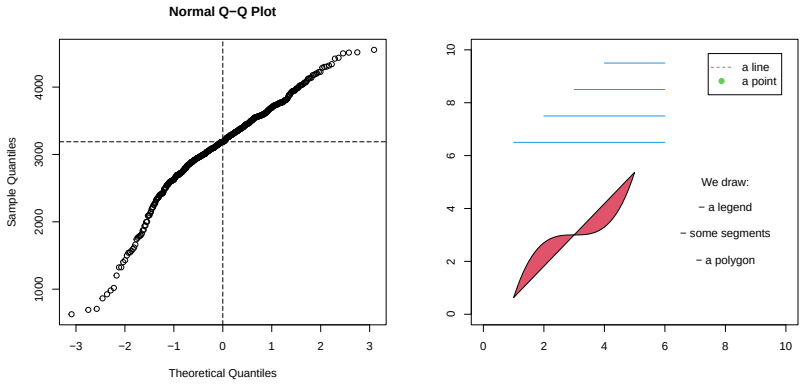

Figure 7: An example with abline() (left) and several other low-level functions (right)

from the variable `bweight` in the data frame `births`, adding the reference normal density curve through the function `curve()`:

```
hist(births$bweight, breaks=20, col=grey(0.9), xlab="Birth weight (gr)", xlim=c(0,6000),
  freq=F, main="An histogram with reference normal density curve")
curve(expr=dnorm(x, mean=mean(births$bweight), sd=sd(births$bweight)), add=TRUE,
  col=4, lwd=2)
```

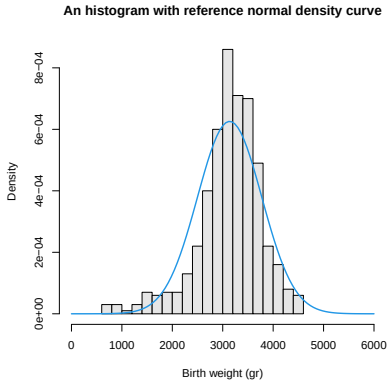

Figure 8: Adding features through the argument 'add'

The argument `freq=FALSE` in `hist()` states that the plot must be represented in probability densities and not in frequencies (note the *y*-axis). Also, we used a light grey colour using `grey()` and an extended range to highlight the curve drawn by `curve()`. This function accepts, as its first argument `expr`, an expression or call in terms of an undefined term `x`, which in this case is the value of the *x*-axis. Here we use the function `dnorm()` to draw the normal density values along `x` for the distribution with parameters equal to the empirical mean and standard deviation of `bweight`.

Special graphs

The graphic capabilities of R are however more extended than how illustrated in the previous sections. Other special plots can easily be created, although we only provide a very brief summary here.

For example, in the traditional graphic system, *multi-panel* plots can be obtained by dividing the inner region into multiple sub-spaces. This can be accomplished by setting to non-default values the graphical

parameters `mfrow` or `mfcol` through `par()`, or more easily and flexibly by using the function `layout()` (or `split.screen()` as a less flexible alternative). These functions divide the inner region into multiple figure regions, each including a plot region. Calls to high-level functions add graphs sequentially in different sub-regions.

Mathematical expressions can be employed in `text()`, `axis()`, `legend()`, or other text-drawing functions by the use of the function `expression()`. The syntax is not trivial, but the help page opened with `help(plotmath)` provides some guidance and lots of examples.

Three-dimensional graphs are easily created in R. The functions `persp()` and `filled.contour()`, amongst others, generate excellent, sophisticated *three-dimensional* plots. Check the help pages for some nice examples.

Several contributed packages offer a wide selection of functions for mapping. In particular, the package MAP provides high-level and low-level functions for drawing maps, along with databases on several countries.

Exporting graphs

Graphs are commonly created in R by plotting them in a screen device, which appears as a sub-window in the standard R GUI, or as the PLOTS window in RStudio. However, these graphs can also be exported outside R in a wide range of formats, without losing their high definition.

The traditional way is to use specific R functions which open an internal *graphic device*, linked with an external file. The calls to different plotting functions add pages to the opened device, until this is closed through the function `dev.off()`. When this happens, the graph is finally transferred to the external file. For instance, in order to save in an external pdf file the box plot created in a previous section, we can use the function `pdf()` with the code:

```
pdf("files/barplot.pdf", width=7, height=7)
plot(ncases ~ agegp, data=esoph)
dev.off()
```

A file BARPLOT.PDF now appears in the folder FILES in the working directory. Of course, different paths can be specified. The arguments `width` and `height` define the dimensions of the figure saved in the external file. Be aware that the plot might appear different than how shown in the screen device.

Various functions can be used to save graphs in different formats, such as `postscript()`, `bmp()`, `jpeg()`, `png()` and `tiff()`. Also, multiple calls to high-level functions create files with multiple pages storing different graphs, at least in the formats which allow that. Check the help pages of each function for details.

Actually, none of the above is required in RStudio. The PLOTS window includes an EXPORT menu that opens a dialogue box for exporting graphs in pdf and other graphical formats. Specifically, you can select the dimensions of the figure, the orientation, the name of the file and the directory, and also preview the final appearance before saving the graph.