University of Glasgow

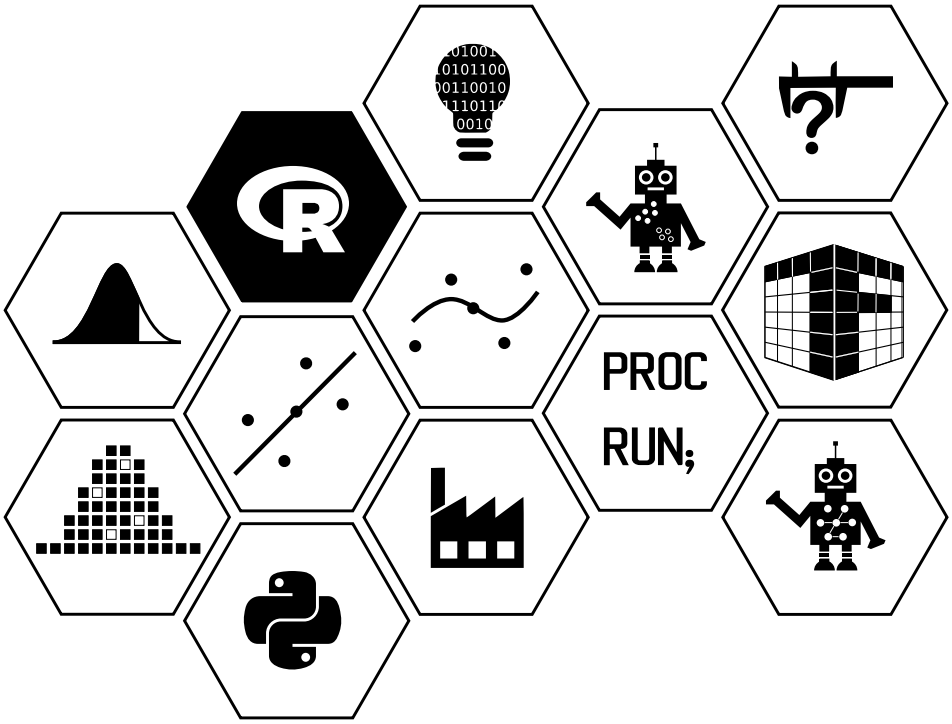
R Programming/ Statistical Computing

Craig Alexander

Academic Year 2023-24

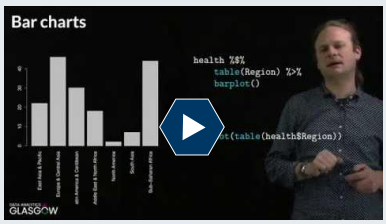
Week 5:

R Graphics



DATA ANALYTICS GLASGOW

Plotting in R



Plotting in R

https://youtu.be/BaZD5uKApdg

Duration: 11m57s

The video gives an overview over the key high-level plotting functions in R. The remaining sections of this chapter explain the underlying functions in more detail. The course *Learning from Data- Data Science Foundations* will look at choosing appropriate statistical plots in more detail.

R has a built-in sophisticated and customisable plotting engine. which we will look at this week.

To get an idea of what R can do, enter

```
demo(graphics)
```

R is widely used for data visualisation: the BBC has been using R to create some of the graphics on its website.

Background reading: Sections 4.2 to 4.4 of Introductory Statistics with R

https://glasgow.summon.serialssolutions.com/#!/search?bookMark=ePnHCXMw42JgAfZbU5khE6Og45_AK9eA1RToWltTEw7YAIgBeN-uMSeDnCdoTTbodNP8okoF0D4ayBHFCqCRSIUgbgYFN9cQZw9dUHMyPb88HjquEZ9k

Chapter 4 of Dalgaard's book explains how to create basic statistical plots in R and how to interpret them. This can be obtained from the library free here

Background reading: Chapter 3 of A First Course in Statistical Programming with R

Chapter 3 explains both the low-level and high-level plotting functions in R and contains a brief discussion of the interpretation of such plots.

Background reading: Chapter 12 of the Introduction to R manual

https://cran.r-project.org/doc/manuals/r-release/R-intro.html#Graphics

The R manual provides a very concise overview of the plotting functions in R. It is a good reference for looking up details such as function and argument names.

R Graph Gallery

http://www.r-graph-gallery.com/all-graphs/

The R Graph Gallery contains many examples of sophisticated plots created using R.

Data stories podcast: Amanda Cox

http://datastori.es/ds-56-amanda-cox-nyt/#t=8:44.342

Amanda Cox was the graphics editor of the New York Times from 2005 to 2016 is a big fan of R.

The function plot

Specifying the coordinates

The function `plot` is at the heart of R's built-in graphics. It can be used to create two-dimensional plots.

The following four commands all create the same scatter plot of `y` against `x`:

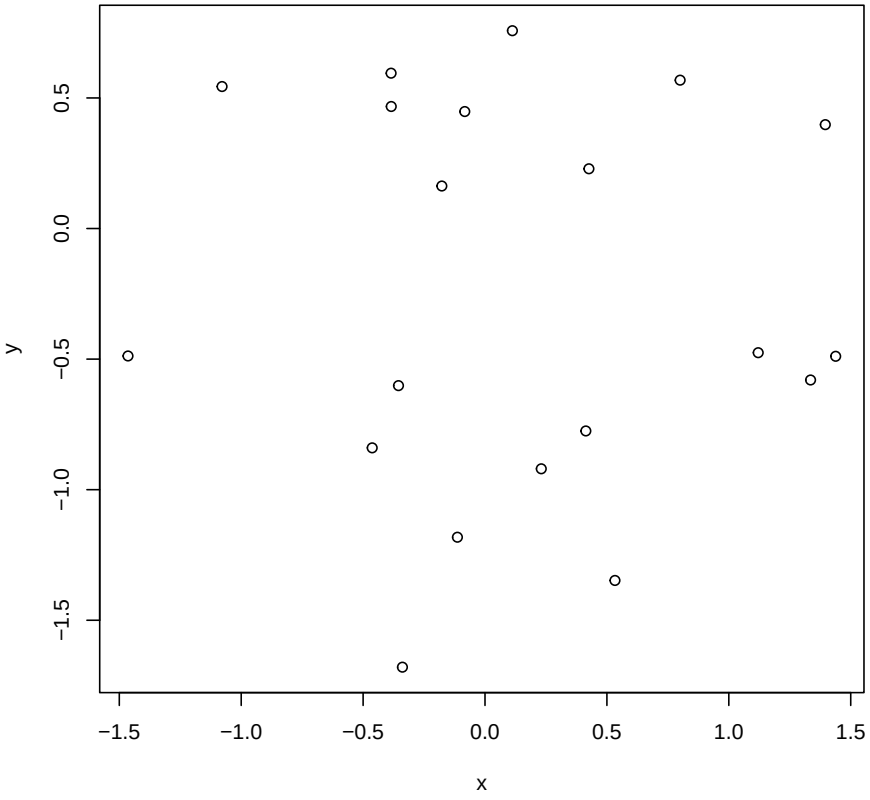
- `plot(x, y, ...)` where `x` and `y` are the vectors containing the coordinates.
- `plot(y~x, ...)` where `x` and `y` are the vectors containing the coordinates, or `plot(y~x, data=data, ...)` where `x` and `y` are columns in `data`.
- `plot(M, ...)`, where `M` is a matrix having two columns (x- and y-coordinates). If `M` has more than two columns, R will ignore these.
- `plot(lst, ...)`, where `lst` is a list with (possibly amongst others) an element `x` and an element `y`.

We first look at a simple example illustrating these three ways of calling `plot`. We first create two vectors `x` and `y` containing white noise.

```
n <- 20
x <- rnorm(n)
y <- rnorm(n)
```

In this example, the simplest way of calling `plot` provides two vectors as arguments:

```
plot(x, y)
```



Alternatively we could have used

```
plot(y~x)
```

(note that `y` comes first if we use the formula interface). If we arrange `x` and `y` in matrix (or data frame or tibble)

```
data <- cbind(x=x, y=y)
```

we can use

```
plot(data)
```

or

```
plot(y~x, data=data)
```

For the former command the order of the columns in `data` matters.

If we store `x` and `y` in a list

```
lst <- list(x=x, y=y)
```

we can use

```
plot(lst)
```

Customising the plot

`plot` has a wide range of optional arguments. The most important ones are:

- `type`: controls how the data is plotted. Use `p` (default) for points, `l` for a line through the points, `b` for a line together with the points, `n` to set up the plot without actually plotting any points. For more options, see `?plot`.
- `xlab`: the label of the x axis (in quotes). Otherwise, R will use the name of the corresponding variable / column.
- `ylab`: the label of the y axis (in quotes). Otherwise, R will use the name of the corresponding variable / column.
- `main`: the title of the plot (in quotes). We can also set the title using the function `title`.
- `sub`: the subtitle of the plot (in quotes).
- `xlim`: if set to `c(xmin, xmax)` the range of the x axis is from `xmin` to `xmax`. If not specified, R will determine the ranges of the axes automatically.
- `ylim`: if set to `c(ymin, ymax)` the range of the y axis is from `ymin` to `ymax`.
- `log`: controls which axes should use a logarithmic scale (`""` (default), `"x"`, `"y"`, or `"xy"`).
- `pch`: the plotting symbol (`0`-`14` for open symbols, `15`-`20` for solid symbols, `21`-`25` for filled symbols (fill colour can be set using `bg`), or a character in quotes).

- `lty`: the type of line (`0`=blank, `1`=solid (default), `2`=dashed, `3`=dotted, `4`=dot-dash, `5`=long dash, `6`=two dashes).
- `lwd`: the width of the line.
- `col`: the colour (either a number, a name in quotes, `#` followed by the hex triplet in quotes (as used in HTML, e.g. `"#ffff00"` for yellow), or the output of the functions such as `rgb`, `hsv`, `gray`/`grey`, `rainbow`). The function `palette` can be used to change how integers map to colours.
- `cex`: the size of the plotting symbol.

If the arguments `col`, `pch`, and `cex` are set to a single value, this value is applied to all points. If they are set to a vector of the same length as the the data points, a different colour / plotting symbol / size is used for each point.

Example 1.

In this example we will plot the health expenditure data which we looked at in the video. It is contained in the RData file

```
load(url("https://github.com/UofGAnalyticsData/R/raw/main/Week%205/w5.RData"))
head(health)
```

```
##     Country                      Region Year Population
## 1   Albania      Europe & Central Asia 2010    3204284
## 2   Algeria Middle East & North Africa 2010   35468208
## 3    Angola         Sub-Saharan Africa 2010   19081912
## 4 Argentina  Latin America & Caribbean 2010   40412376
## 5   Armenia      Europe & Central Asia 2010    3092072
## 6 Australia        East Asia & Pacific 2010   22065300
##   LifeExpectancy HealthExpenditure
## 1       76.90095          220.2286
## 2       72.85254          198.1556
## 3       50.65366          146.1099
## 4       75.63215          759.2994
## 5       73.78356          133.7856
## 6       81.69512         5173.5024
```

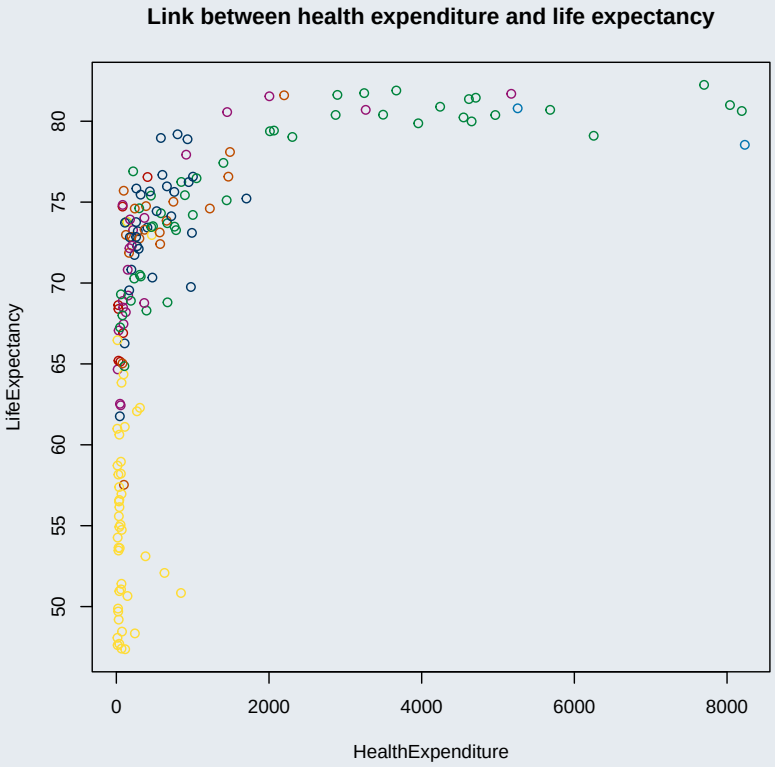
We can create a scatter plot of life expectancy against health expenditure using

```
plot(LifeExpectancy~HealthExpenditure, data=health)
```

We can now use `col` to use colour to denote the geographical region. However, `Region` is a factor, so we first need to convert it to integers (which R accepts as colours). We can do this using the function

`unclass` – we add 1 to the result, otherwise one group would have their points drawn in black (R's first choice of colour).

```
plot(LifeExpectancy~HealthExpenditure, data=health, col=1+unclass(Region))
title("Link between health expenditure and life expectancy")
```



We should now add a legend to the plot. We will look at this later on.

The exposition pipe operator from magrittr

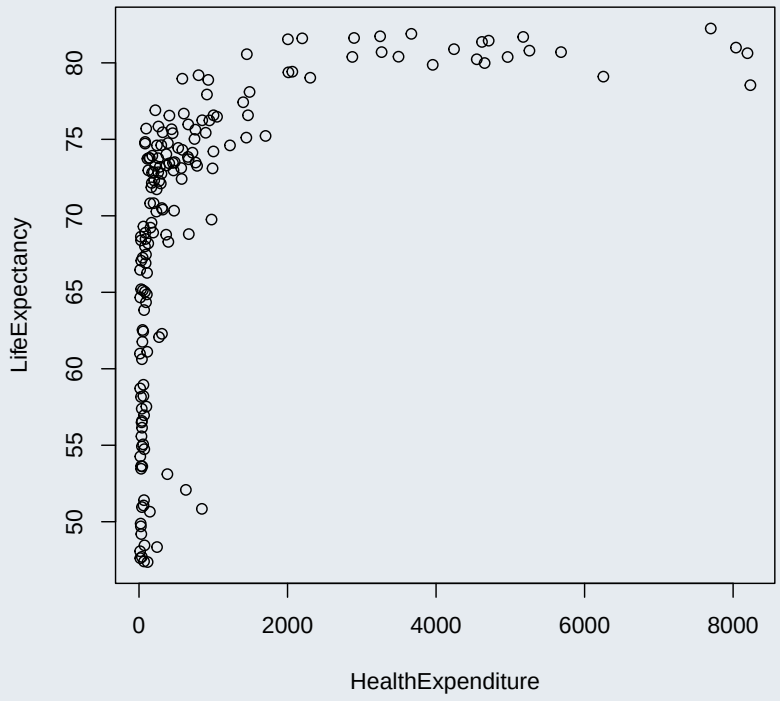
In the above example we have avoided having to use `health\$` in front of the variables by using the additional argument `data=health`. However not all plotting functions in R allow for a `data` argument.

Another strategy of avoiding having to use `health$` is to use `attach`:

```
attach(health)
```

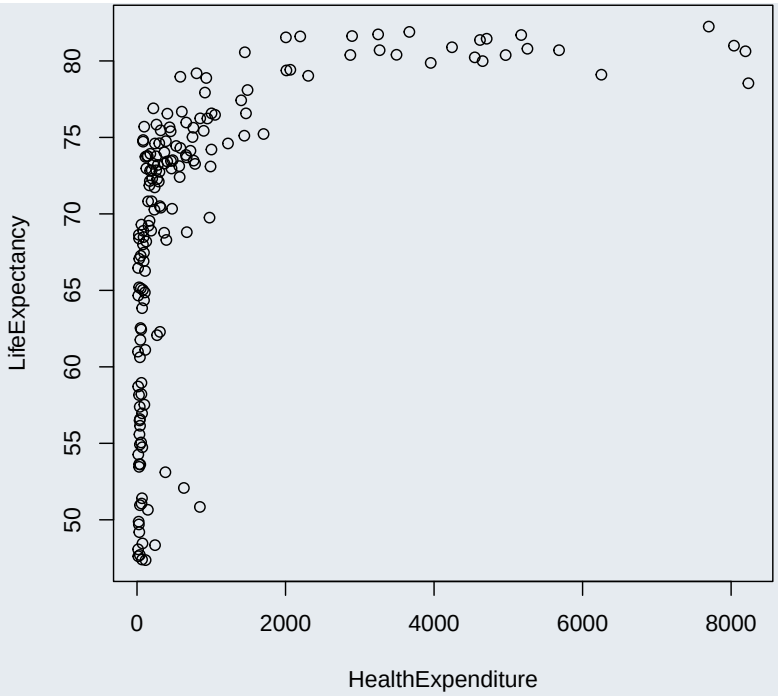
```
## The following objects are masked from health (pos = 3):
##
##     Country, HealthExpenditure, LifeExpectancy,
##     Population, Region, Year

## The following objects are masked from health (pos = 4):
##
##     Country, HealthExpenditure, LifeExpectancy,
##     Population, Region, Year

## The following objects are masked from health (pos = 5):
##
##     Country, HealthExpenditure, LifeExpectancy,
##     Population, Region, Year

## The following objects are masked from health (pos = 6):
##
##     Country, HealthExpenditure, LifeExpectancy,
##     Population, Region, Year

## The following objects are masked from health (pos = 7):
##
```

```
##     Country, HealthExpenditure, LifeExpectancy,
##     Population, Region, Year

## The following objects are masked from health (pos = 8):
##
##     Country, HealthExpenditure, LifeExpectancy,
##     Population, Region, Year

## The following objects are masked from health (pos = 9):
##
##     Country, HealthExpenditure, LifeExpectancy,
##     Population, Region, Year

## The following objects are masked from health (pos = 10):
##
##     Country, HealthExpenditure, LifeExpectancy,
##     Population, Region, Year

## The following objects are masked from health (pos = 13):
##
##     Country, HealthExpenditure, LifeExpectancy,
##     Population, Region, Year

plot(HealthExpenditure, LifeExpectancy)
```



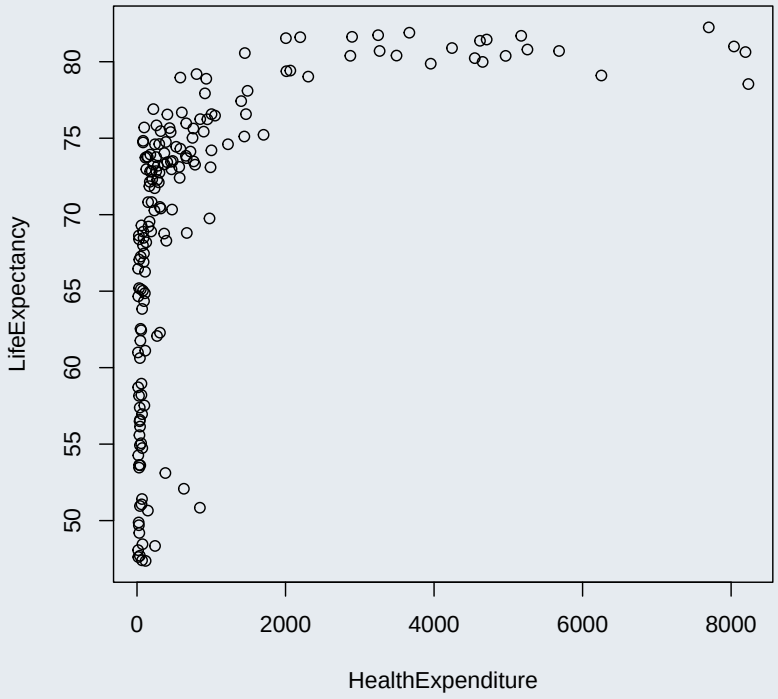
However, we have seen that `attach` can be a little tricky if we make changes to the data set, so we better avoid using it. The function `with` provides a (slightly clunky) alternative:

```
with(health, plot(HealthExpenditure, LifeExpectancy))
```



The exposition pipe operator %$% from magrittr, lets us write this a bit more elegantly:

```
library(magrittr)
health %$% plot(HealthExpenditure, LifeExpectancy)
```



Task 1.

The package `ggplot2` contains a data set `diamonds` containing the prices and other attributes of almost 54,000 diamonds.

```
library(ggplot2)
diamonds <- as.data.frame(diamonds)
head(diamonds)
```

```
##   carat       cut color clarity depth table price    x    y
```

```
## 1  0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98
## 2  0.21   Premium     E     SI1  59.8    61   326 3.89 3.84
## 3  0.23      Good     E     VS1  56.9    65   327 4.05 4.07
## 4  0.29   Premium     I     VS2  62.4    58   334 4.20 4.23
## 5  0.31      Good     J     SI2  63.3    58   335 4.34 4.35
## 6  0.24 Very Good     J    VVS2  62.8    57   336 3.94 3.96
##       z
## 1 2.43
## 2 2.31
## 3 2.31
## 4 2.63
## 5 2.75
## 6 2.48
```

Create a scatter plot of carat against price, using different colours to denote the different colour and different plotting symbols to denote the different cuts.

Plotting objects

Many R objects (such as model fits) have a `plot` method, which draws a visualisation of (or diagnostic check relating to) this object.

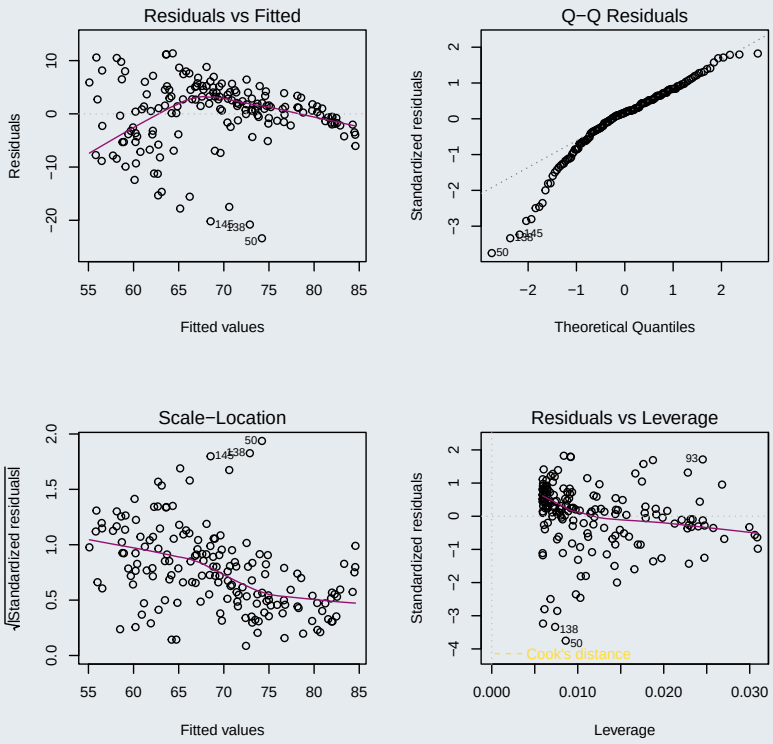
Example 2.

Suppose we fit a linear regression model to the data from the example:

```
model <- lm(LifeExpectancy ~ log(HealthExpenditure), data=health)
```

The `plot` method for linear model objects produces a series of diagnostic plots.

```
plot(model)
```



You will learn more about linear regression models and also how to interpret these diagnostic plots in *Predictive Modelling*.

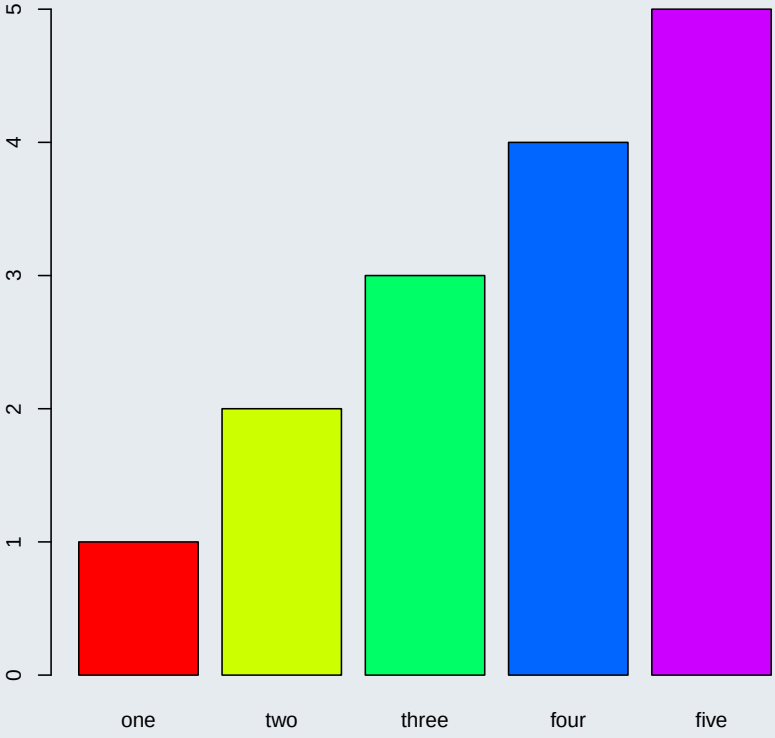
High-level functions for statistical plots

Bar charts and pie charts

The function `barplot(height, ...)` can be used to create bar plots. The vector `height` hereby contains the heights of the bars.

Example 3.

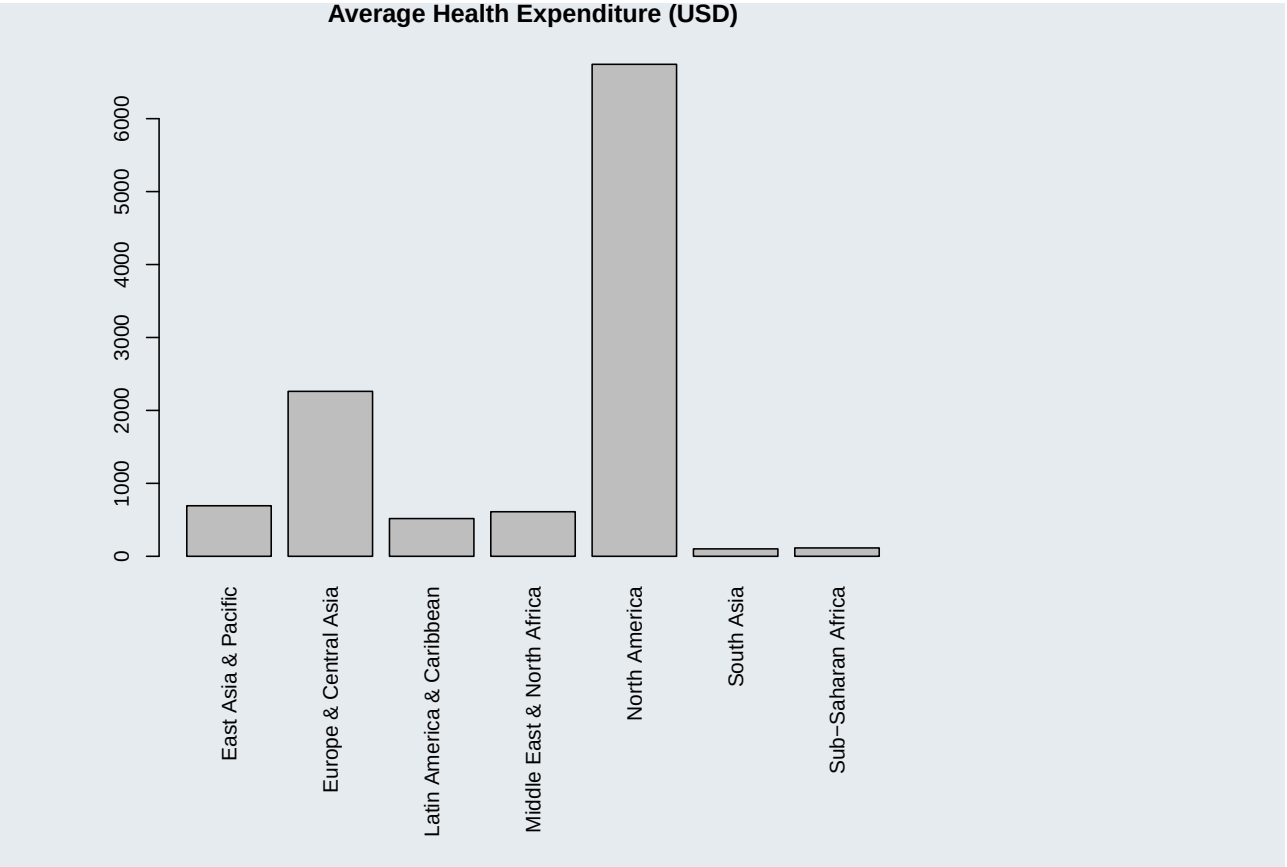
```
height <- 1:5
names(height) = c("one", "two", "three", "four", "five")
barplot(height, col=rainbow(5))
```



Example 4.

We can draw a bar plot of the average health expenditure in the example data set using the following R code.

```
library(dplyr)
HESummary <- health %>%                                    # Get avg health exp
                group_by(Region) %>%
                summarise(HealthExpenditure=mean(HealthExpenditure))
x <- HESummary$HealthExpenditure                           # Turn into named vector
names(x) <- HESummary$Region
par(las=3, mar=c(12.1, 4.1, 4.1, 2.1))                     # Make space for labels
barplot(x)                                                 # Create bar plot
title("Average Health Expenditure (USD)")
```

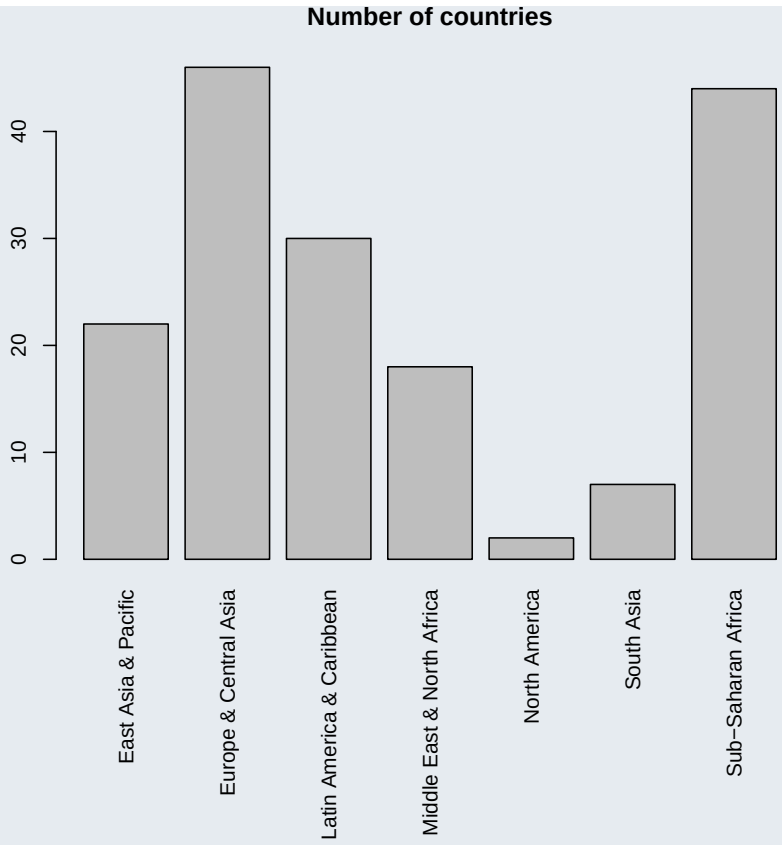


If we want to use the function `barplot` to chart frequencies of categorical variables, we first need to tabulate these using the function `table`.

Example 5.

We can create a barchart of the number of countries in each region using

```
par(las=3, mar=c(12.1, 4.1, 4.1, 2.1))                    # Make space for labels
health %$%
    table(Region) %>%
    barplot()
title("Number of countries")
```



We have used `magrittr`'s pipe exposition operator %$%. We could have also used

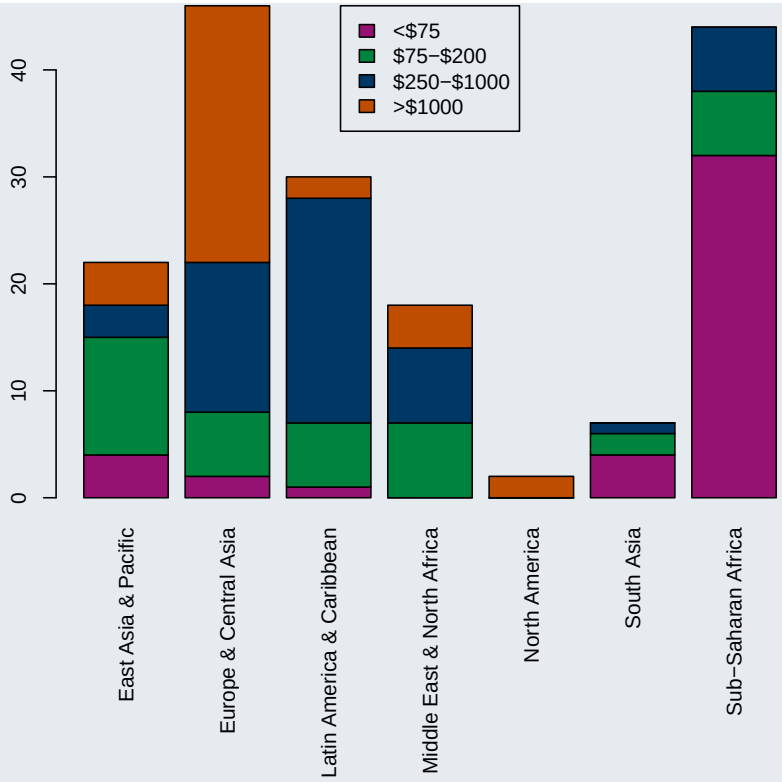
```
barplot(table(health$Region))
```

If the argument `height` to `barplot` is a matrix the bars are shown in groups. If we use the additional argument `beside=FALSE`, the bars are stacked.

Example 6.

Let's use this to illustrate the distribution of health expenditure in the different regions. We start by discretising the health expenditure and then plotting the number of countries for each level and region.

```
par(las=3, mar=c(12.1, 4.1, 4.1, 2.1))      # Increase space for labels
health %>%
    mutate(HealthExpenditure=cut(HealthExpenditure,
                                 breaks=c(0,75,250,1000,10000))) %>%
    select(HealthExpenditure, Region) %>%
    table() %>%
    barplot(col=2:5)
legend("top", fill=2:5, c("<$75","$75-$200","$250-$1000",">$1000"))
```

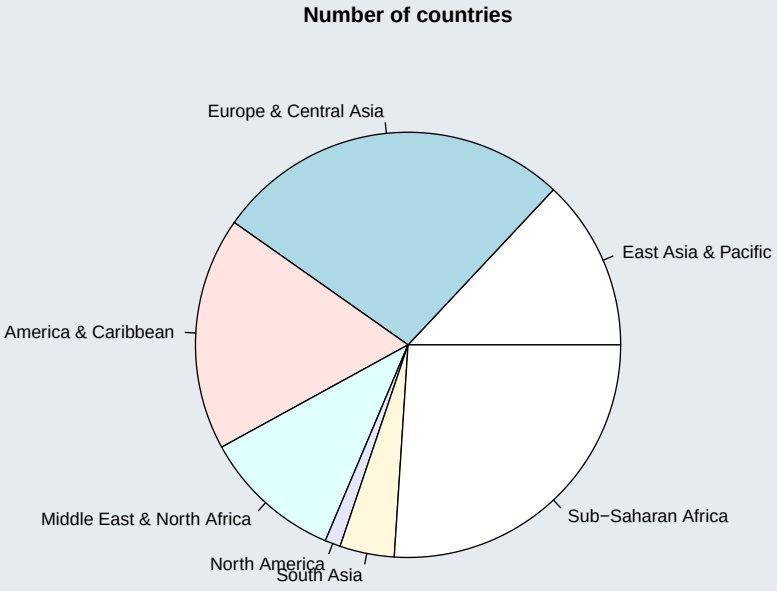


We can use the function `pie(x,...)` to draw a pie chart of the proportions given by the vector `x` (which will be renormalised, if necessary).

Example 7.

We can draw a pie chart showing the proportions of countries which comes from each region using the following R code.

```
health %$%
    table(Region) %>%
    pie()
title("Number of countries")
```



Note that pie charts are not ideal: the human eye is better at judging lengths than angles and pie charts are essentially just about judging angles.

Boxplots

Boxplots can be created using the function `boxplot`.

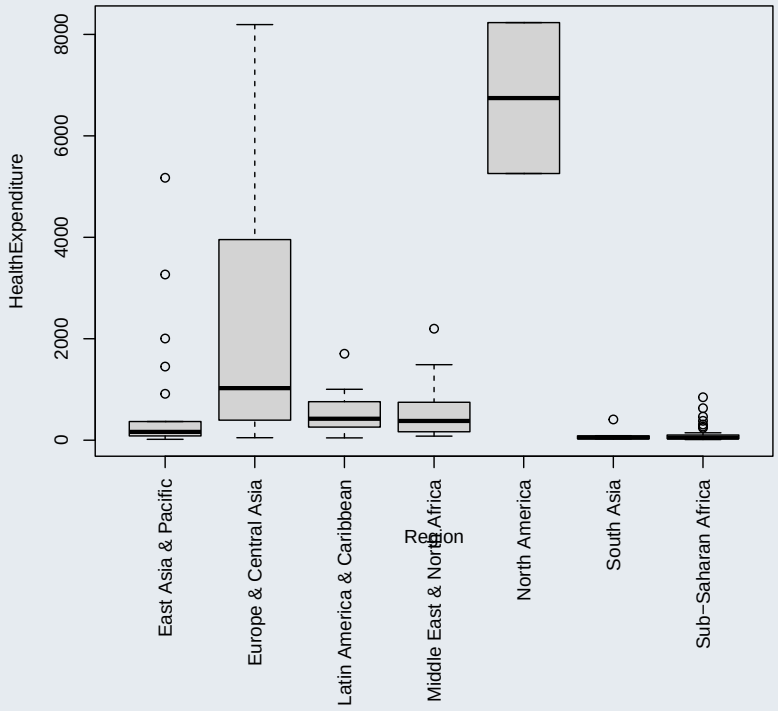
`boxplot(y, ...)` creates a box plot of the data in the vector `y`. If `y` is a data frame then R will draw one box plot per column (using a common y axis).

If `x` is a categorical variable then `boxplot(y~x, ...)` draws a boxplots separately for each level of `x`.

Example 8.

We draw boxplots showing the distribution of the health expenditure in each region using the following R code.

```
par(las=3, mar=c(12.1, 4.1, 4.1, 2.1))            # Increase space for labels
health %$% boxplot(HealthExpenditure ~ Region)
```



Task 2.

Let's return to the diamonds data from task 1. Create boxplots of the prices of diamonds as a function of cut and colour.

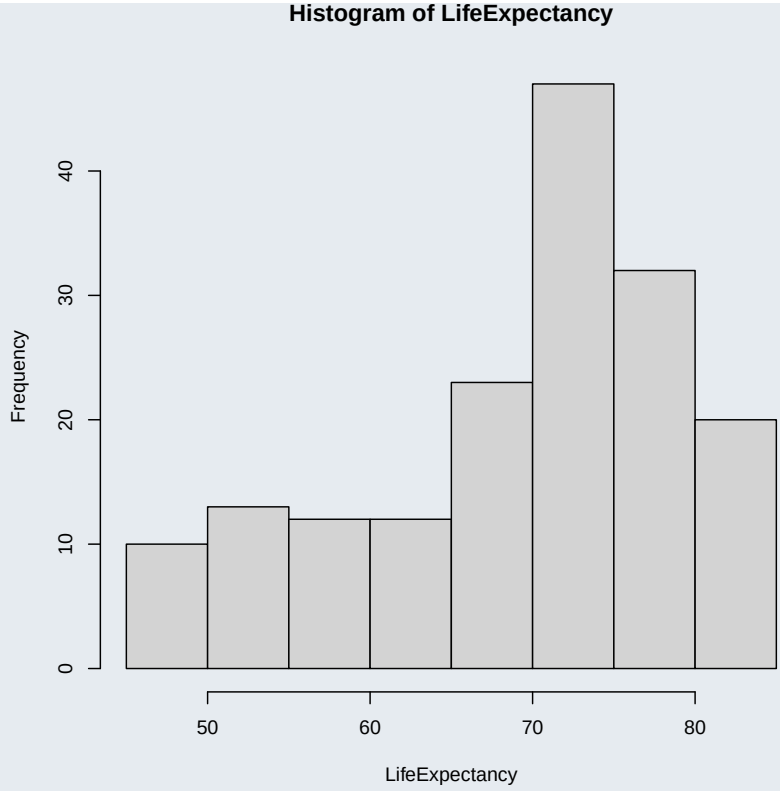
Histograms and density plots

Histograms can be created using the function `hist(x, ...)`.

Example 9.

A histogram of health expenditure can be created using …

```
health %$% hist(LifeExpectancy)
```



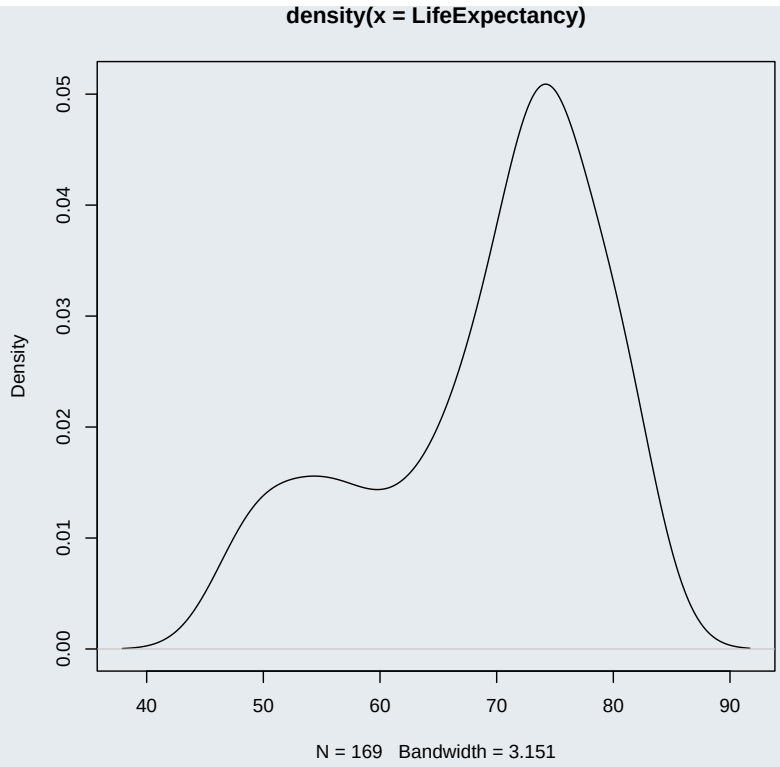
R automatically chooses the breakpoints. You can use the argument `breaks` to override it: `breaks=n` forces R to use `n` breakpoints, alternatively you can set `breaks=vec`, where `vec` is a vector containing the breakpoints to be used.

A histogram is a discrete approximation to the probability density function. You can estimate the probability density function using the function `density`:

Example 10.

A plot of the (estimated) density of health expenditure can be created using …

```
health %$% plot(density(LifeExpectancy))
```



Scatterplot matrices

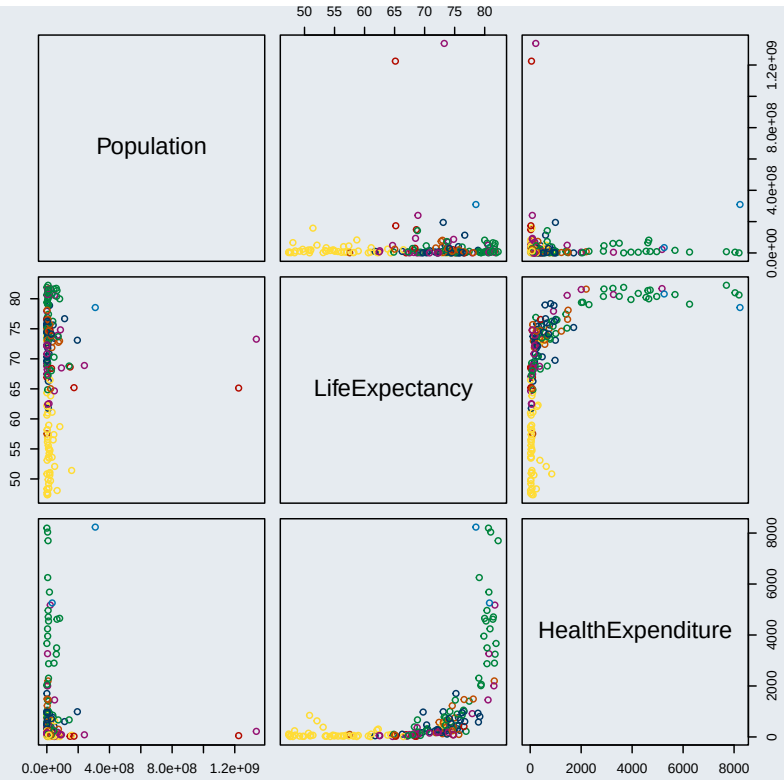
We can use the function `plot` to create a scatterplot of two continuous variables. Often, data sets contain more than just two continuous variables. The function `pairs` draws a scatterplot matrix. It contains a scatter plot of every variable against every other variable.

Colour (and plotting symbols, ...) can be set in the same way as for plot.

Example 11.

The fourth to sixth column of the health expenditure data contain continuous data, which we can visualise using a scatterplot matrix.

```
pairs(health[,4:6],
      col=1+unclass(health$Region))
```

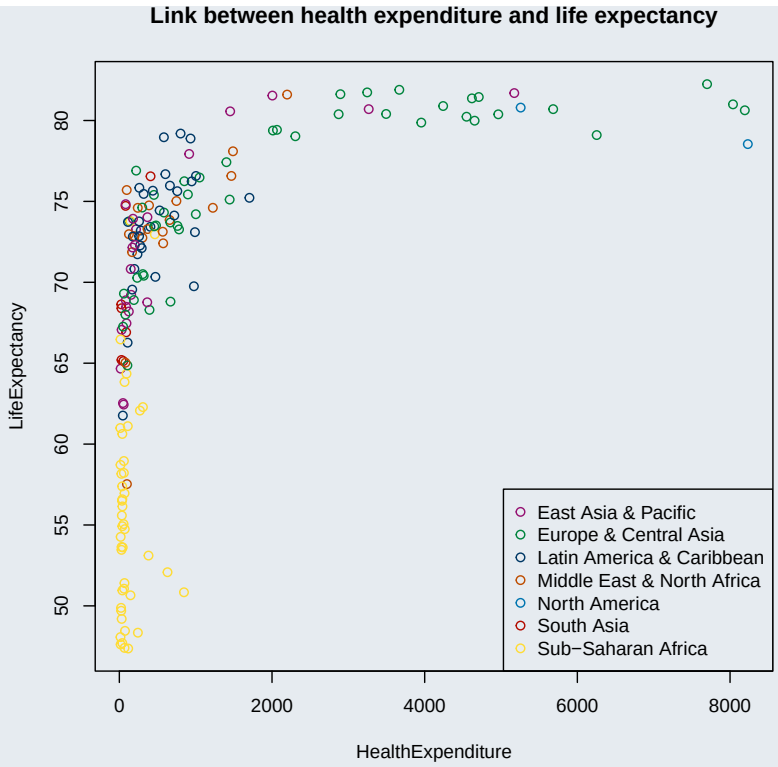


Legends

The function `legend(position, type=values, legend=legend)` can be used to add a legend to a plot. More than one `type=values` expression can be used. `legend` is the vector containing the labels to be used in the legend. `position` can be `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"`. Alternatively you can specify the coordinates of the legend.

Example 12.

We will now add a legend to our plot of the health expenditure from example 1.




```
health %$%
    legend("bottomright",  pch=1, col=1+1:nlevels(Region), legend=levels(Region))
```

`pch=1` is needed to make sure that R uses the standard plotting symbol in the legend.

Low-level plotting functions

Adding to plots using points and lines

The function `lines` and `points` can be used to add lines or points to an existing plot. `lines` behaves like `plot` using `type="l"`, and `points` behaves like `plot` using `type="p"`, except that the points/lines are added to the active plot, rather than a new plot. `points` and `lines` can be used the same way as `plot` (except for the arguments `title`, `sub`, `xlim`, `ylim`, `log`, `type`, which cannot be used).

Example 13.

Suppose we want to highlight the observations belonging to the UK, the US and Australia in the plot of health expenditure and life expectancy.

We start with redrawing the plot

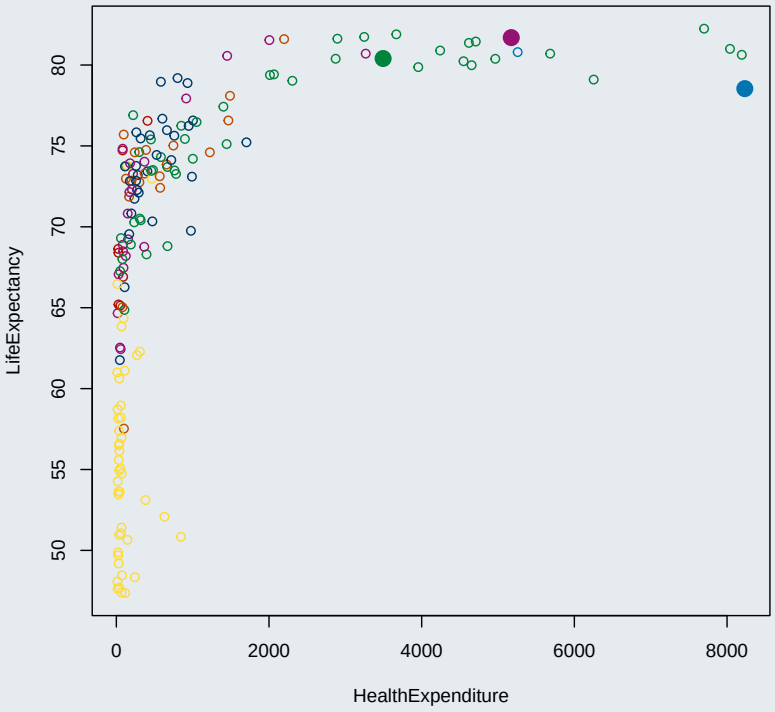
```
plot(LifeExpectancy~HealthExpenditure, data=health, col=1+unclass(Region))
```

Then we can extract the data belonging to these three countries …

```
health2 <- health %>%
    filter(Country %in% c("Australia", "United Kingdom", "United States"))
```

… and we can then draw the points for these three countries with a filled circle (`pch=16`) and twice the size (`cex=2`):

```
health2 %$%
    points(HealthExpenditure, LifeExpectancy, col=1+unclass(Region), pch=16, cex=2)
```



Task 3.

Consider two vectors `x` and `y` created using

```
n <- 1e3
x <- runif(n, 0, 2*pi)                  # x is random uniform from (0,2*pi)
x <- sort(x)                            # Sort entries in x to avoid a mess
y <- sin(x)                             # Set y to the sine of x
y.noisy <- y + .25 * rnorm(n)           # Create noisy version of y
```

Consider the following two blocks of code:

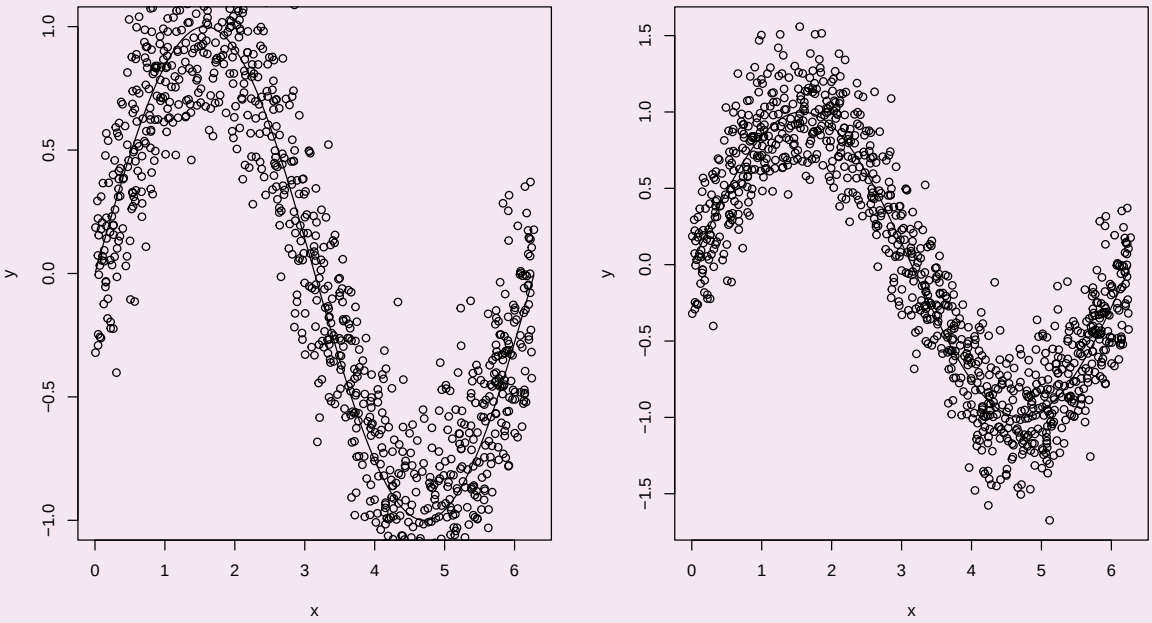
```
plot(x, y.noisy, ylab="y")
lines(x, y, col=2)
```

and

```
plot(x, y, type="l", col=2)
points(x, y.noisy)
```

The two plots generated are shown below (in arbitrary order).

```
par(mfrow=1:2)
plot(x, y, type="l")
points(x, y.noisy)
plot(x, y.noisy, ylab="y")
lines(x, y)
```



What is the difference between the two commands? Why do the plots looks different? Which plot comes from which command?

The function `abline` can be used to add a straight line to an existing plot:

- `abline(h=ypos, ...)` draws a horizontal line at `ypos`.
- `abline(v=xpos, ...)` draws a vertical line at `xpos`.
- `abline(a=intercept, b=slope, ...)` draws a line with `intercept` as its intercept and `slope` as its slope.

`col`, `lwd` and `lty` can be used as additional arguments.

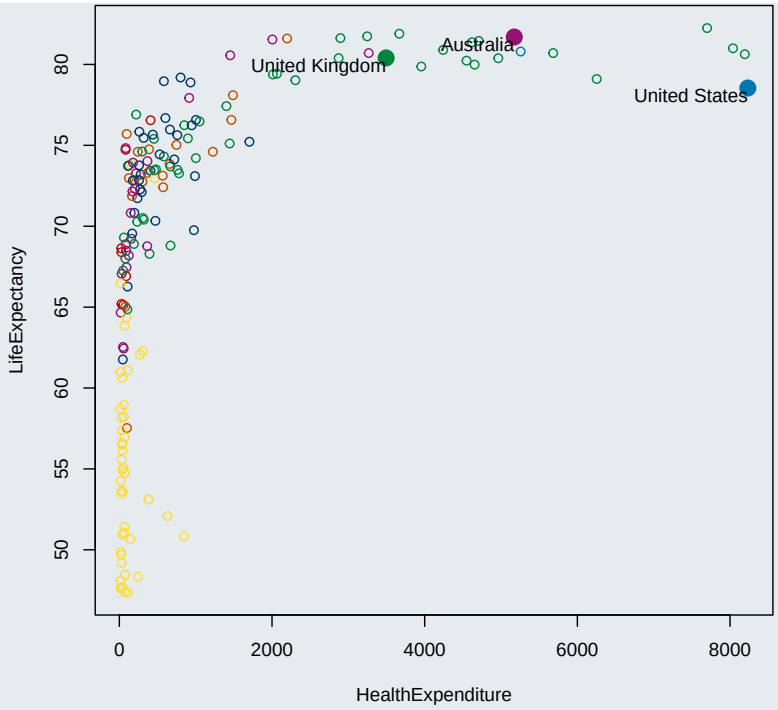
Adding text

The function `text(x, y, text, ...)` plots the text `text` (one character string or a vector of strings) at the co-ordinate(s) (`x`, `y`). The optional arguments include `col`, `cex`, and `adj=c(horiz, vert)`, which sets the horizontal adjustment to `horiz` (`0`: left justified, `0.5` centred, `1`: right justified) and the vertical adjustment to `vert` (`0`: bottom, `0.5` centre, `1`: top). The default is all centred `c(0.5, 0.5)`.

Example 14.

Suppose we now want to label the observations belonging to the US, UK and Australia in the plot from example 13.

```
health2 %$%
    text(HealthExpenditure, LifeExpectancy, Country, adj=c(1,1))
```



`adj=c(1,1)` gets the text to be "attached" to the coordinate systems at the top-right of each text label.

Drawing rectangles and polygons

The functions `rect` and `polygon` can be used to draw filled rectangles and polygons.

`rect(xleft, ybottom, xright, ytop, ...)` draws a rectangle with bottom left corner (`xleft`, `ybottom`) and top right corner (`xright`, `ytop`).

`polygon(x, y, ...)` draws a polygon the with vertices in (`x`, `y`).

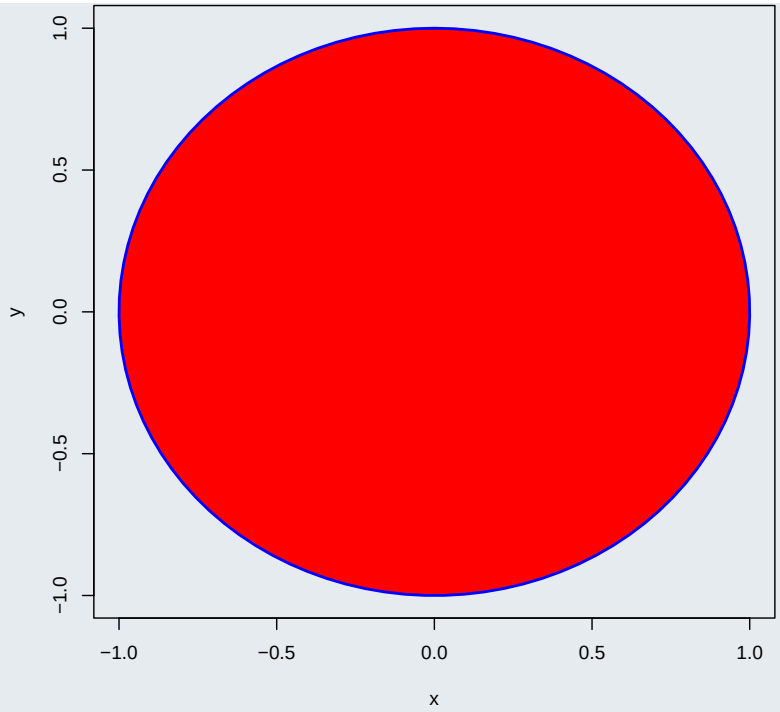
The following optional arguments can be used:

- `border`: the colour of the outline. Use `border=NA` to draw no outline.
- `lwd` / `lty`: the line width/line type of the outline.
- `col`: the colour used for filling the polygon. Use `col=NA` for a transparent rectangle/polygon (i.e. with no fill).
- `density`: the density of shading lines (defaults to `NULL`, i.e no shading lines, but a solid fill is used).

Example 15.

The following R code draws a red circle of radius 1 having a blue outline. Note the use of `plot` with argument `type=n` to set up the plot (coordinate axes, etc.).

```
t <- seq(0, 2*pi, length.out=100)
circle <- cbind(sin(t), cos(t))
plot(circle, type="n", xlab="x", ylab="y")
polygon(circle, col="red", border="blue", lwd=2)
```



The circle looks more like an oval. In order to obtain a plot with perfectly equal scales (so that the circle looks like a circle) one can use the function `eqscplot` from the package `MASS` instead of `plot`.

3D and image plots

The functions `persp(x, y, M, ...)`, `image(x, y, M, ...)`, `contour(x, y, M, ...)`, and `filled.contour(x, y, M, ...)` can be used to visualise functions $f(x, y)$ of two variables x and y. The functions are also useful when visualising spatial or map data.

The two (optional) arguments are vectors containing the different values of `x` and `y` respectively. `M` is a matrix containing the values of f, such that

$$M_{ij} = f(x_i, y_j)$$

In other words, these functions need the data in "wide" format.

Example 16 (Bivariate Gaussian density).

In this example we will plot the probability density function of the two-dimensional standard normal distribution

$$f(x, y) = \phi(x) \cdot \phi(y),$$

which is the product of the probability density function of the univariate standard normal distribution (as X and Y are independent).

We need to evaluate the function $f(x, y)$ for all combinations of input values x_i and y_j. It is easiest if we start with putting together a data frame which contains all these combinations: this can be done using the function `expand.grid`:

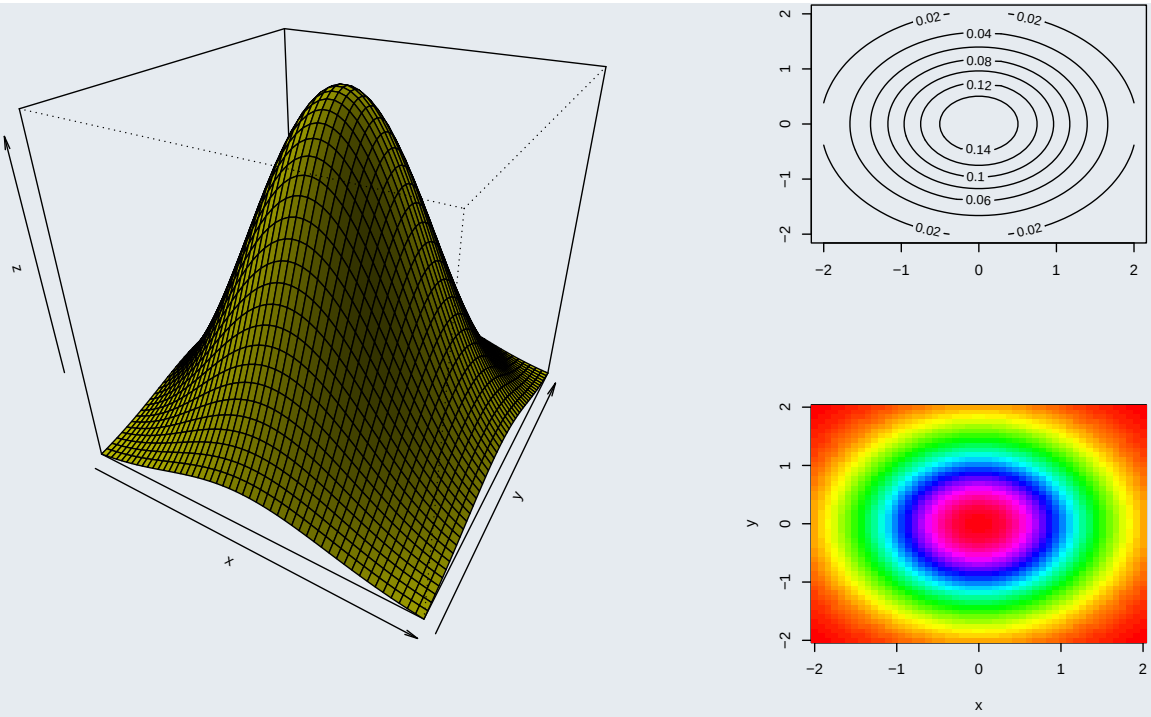
```
library(magrittr)
library(dplyr)
x <- seq(-2, 2, len=50)
y <- seq(-2, 2, len=50)
data <- expand.grid(x=x, y=y) %>%
    mutate(z=dnorm(x)*dnorm(y))
```

We next have to use `spread` to arrange the data in wide matrix form.

```
library(tidyr)
z <- data %>%
       spread(y, z) %>%
       select(-x) %>%
       as.matrix()
```

We can then create the 3D and image plots using

```
layout(rbind(c(3, 3, 1),
             c(3, 3, 2)))                          # Split the figure
contour(x, y, z)                                   # Contour plot
image(x, y, z, col=rainbow(100))                   # Image plot
persp(x, y, z, theta=30, phi=30, col="yellow", shade=0.5) # 3D perspective plot
```



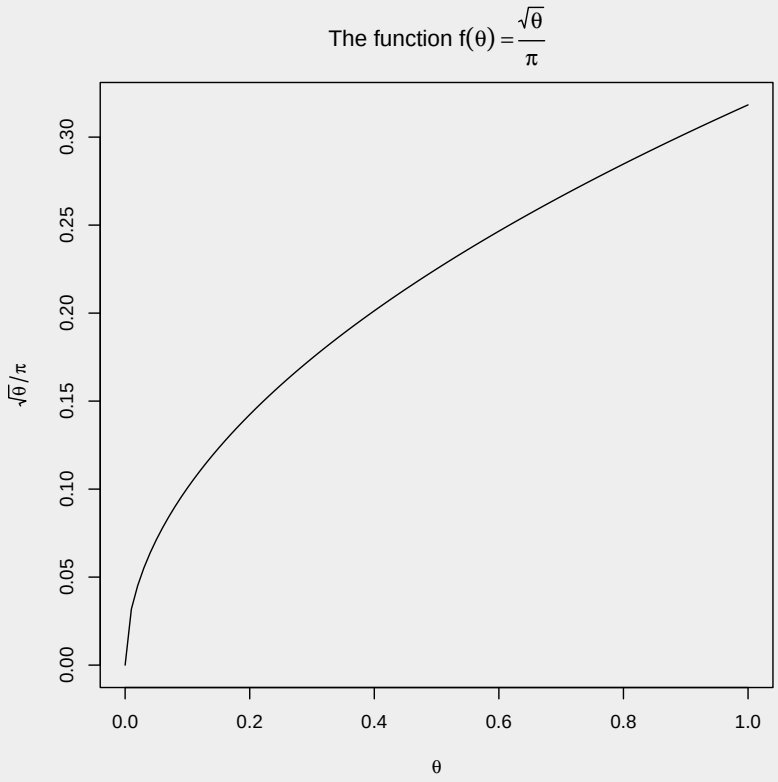
The arguments `theta` and `phi` of the `persp` allow for changing the angle from which the surface is viewed.

Mathematical notation in plots

Supplementary material:
Using mathematical symbols and formulae in plots

The text arguments used to display text in graphics (`xlab`, `ylab`, `main`, `sub`, ...) and the function `text` can be used as well to typeset mathematical formulae in a TeX-like manner. The formulae are not enclosed in quotation marks, but given as an argument to the functions `quote` or `expression`, as the following example shows:

```
theta <- seq(0, 1, by=0.01)
plot(theta, sqrt(theta)/pi, type="l",
     xlab=quote(theta), ylab=quote(sqrt(theta)/pi),
     main=quote("The function "*f(theta)==frac(sqrt(theta),pi)))
```



Note that * is used to juxtapose two expressions. == gives a = `frac(,)` gives $\frac{x}{y}$. Use the Latin transliteration of Greek letters (e.g. `alpha`). See `?plotmath` for details.

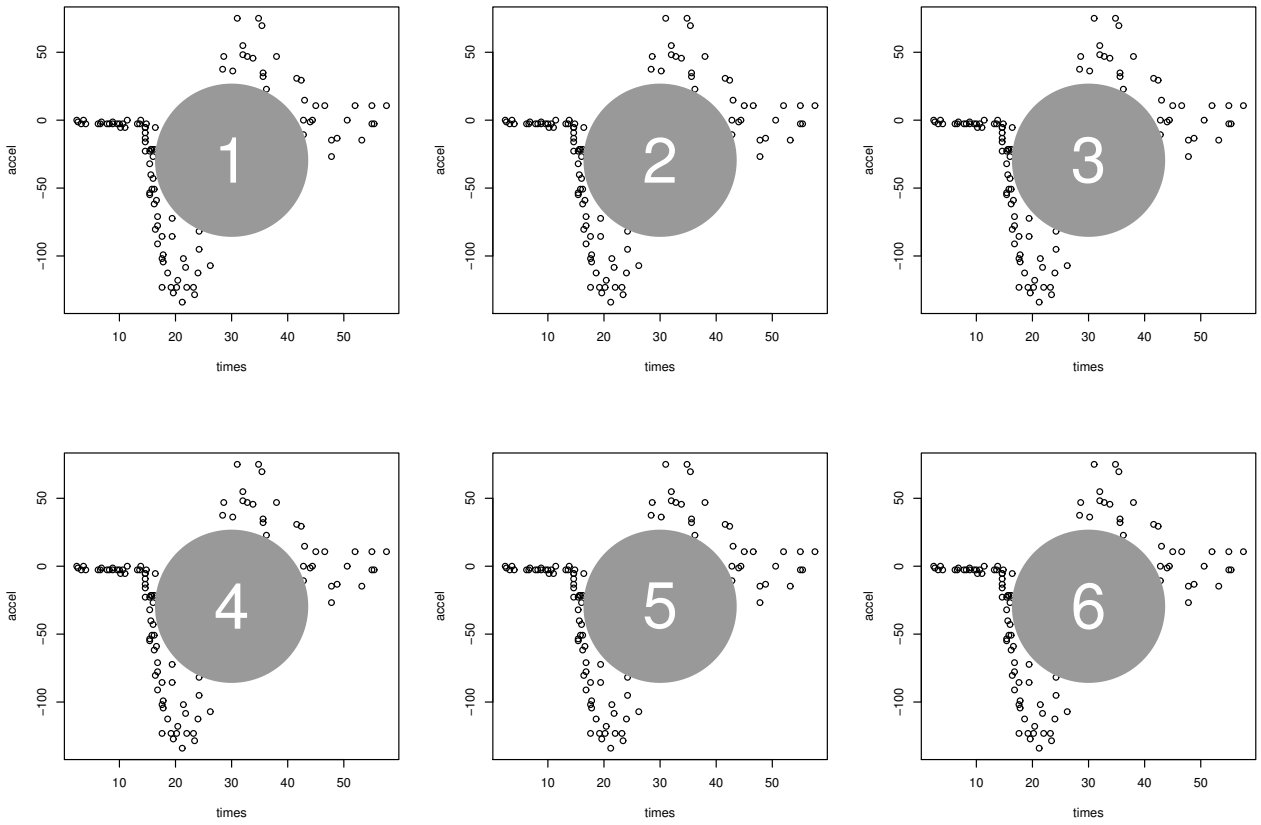


Figure 1: Figure split using `mfrow=c(2,3)`

Setting plot preferences

The function `par` can be used to customise plots in many ways (see `?par`). We have used `par` already to adjust the margins and to change the orientation of the tickmarks. Another important use of `par` is to put more than one plot onto a figure.

Margins

The margins of the plot can be changed by calling `par(mar=c(bottom, left, top, right))` before plotting.

Arranging plots on a grid

`par(mfrow=c(nrows, ncols))` divides the figure into `nrows` rows and `ncols` columns, which will be used in *row-wise* order.

`par(mfcol=c(nrows, ncols))` divides the figure into `nrows` rows and `ncols` columns, which will be used in *column-wise* order.

Task 4.

Create a plot of $\sin(x)$ and $\cos(x)$ right next to each other for $x \in (0, 2\pi)$.

Sophisticated arrangement of plots using layout

For more sophisticated arrangements you can use the function `layout`. The first (and possibly only) argument of `layout` is an integer matrix describing how the figure is to be divided. For example,

```
layout.matrix <- rbind(c(1, 1, 2),
                       c(1, 1, 3))
layout(layout.matrix)
```

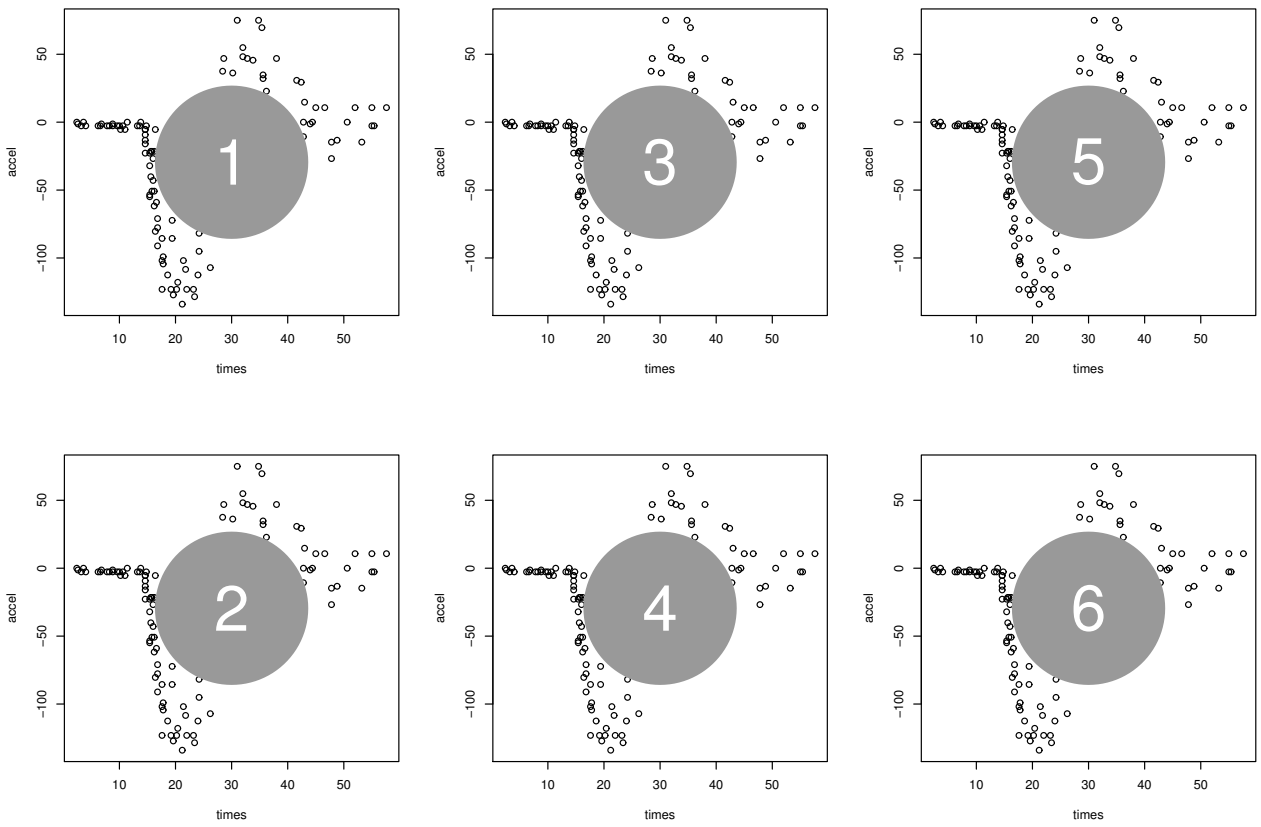


Figure 2: Figure split using `mfcol=c(2,3)`

creates the layout shown in the figure below.

The optional arguments `widths` and `heights` can be used to change the widths of the columns and heights of the rows. If they are not specified, each column / row has the same width / height.

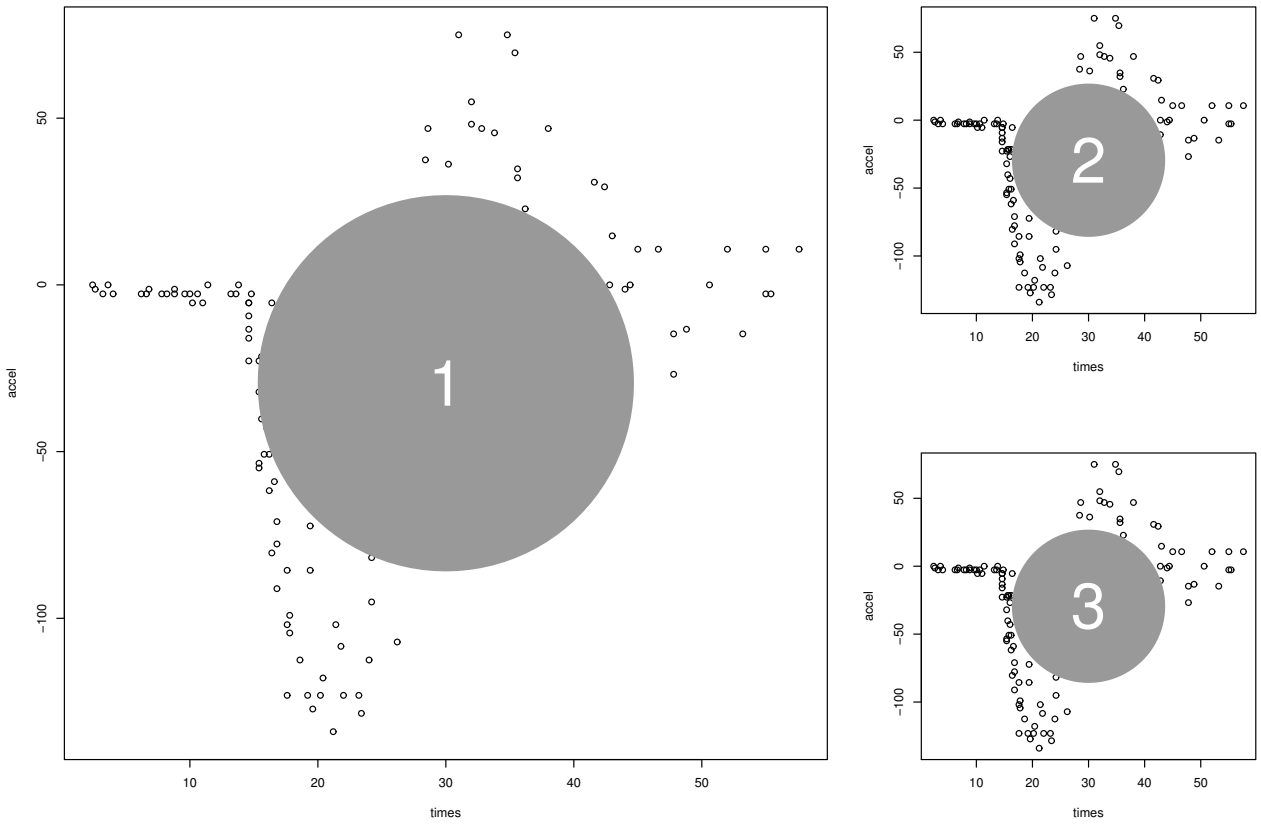


Figure 3: A plot region split using `layout`

Exporting plots

By default R uses the screen as plotting device.

- To copy your graphics from the classical R GUI into another application under Windows, choose *File* > *Copy to the clipboard* and then *as a Metafile*. You can then paste the graphics into your document (in Word, Powerpoint, …).
- In RStudio, to copy a plot to the clipboard choose *Export* and then *Copy plot to clipboard* …. Choosing the *Metafile* option usually gives better quality results. You can also export the plot to a variety of other file formats.

The screen is not the only graphics device under R and, especially if you want to create a large number of plots, it is simpler to use a dedicated graphics device, rather than using the GUI to export the active graphics device. The most important graphics devices are:

- `win.metafile("filename.emf" , width=w, height=h)` creates an enhanced Windows metafile with the dimensions `w`×`h` (not available on Mac or Linux).
- `pdf("filename.pdf", width=w, height=h)` and `postscript("filename.ps", width=w, height=h)` create a PDF or PostScript file with the dimensions `w`×`h`.
- `svg("filename.svg", width=w, height=h)` creates an SVG file with the dimensions `w`×`h`.
- `png("filename.png", width=w, height=h)` and `jpeg("filename.jpg", width=w, height=h)` create an PNG / JPEG image of resolution `w`×`h` pixels.

The `w` and `h` arguments are optional.

When using any of the above graphics devices, you have to use `dev.off()` to close the device once you have finished the plot, otherwise the file remains open (and most likely unfinished).

More examples

Example 17 (Fisher's iris data).

In this example we look at Fisher's famous iris data, a data set available in R. The data gives, for 50 flowers, four measurements (sepal length and width and petal length and width). The flowers in the data set come from three species: Iris setosa, versicolor, and virginica.

```
library(magrittr)
library(dplyr)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

We will start with putting together a plot of the sepal length and sepal width. We will use the plotting symbol and colour to denote the species.

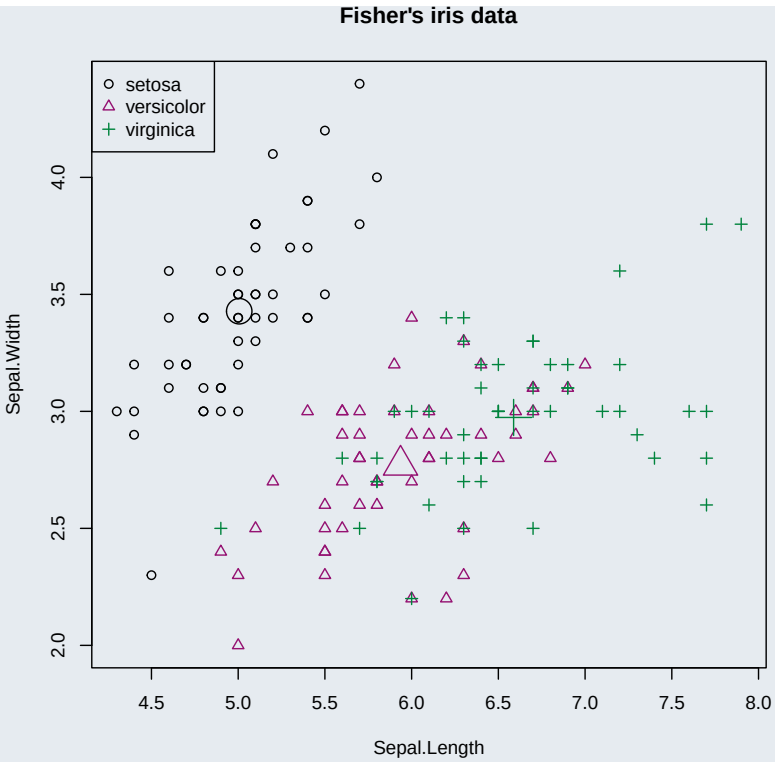
```
plot(Sepal.Width~Sepal.Length, data=iris,
     col=unclass(Species),
     pch=unclass(Species),
     main="Fisher's iris data")
```

Next, we will add a legend.

```
legend("topleft", pch=1:3, col=1:3,
       legend=c("setosa", "versicolor", "virginica"))
```

Finally, we will use a large plotting symbol to denote the mean values for each species

```
species.means <- iris %>%
                    group_by(Species) %>%
                    summarise_all(mean)
species.means %$%
    points(Sepal.Length, Sepal.Width, pch=1:3, col=1:3, cex=3)
```



Example 18 (House prices in the UK).

Once you have loaded

```
load(url("https://github.com/UofGAnalyticsData/R/raw/main/Week%205/w5.RData"))
```

you can access a data frame `hp`, which contains the median house price for all regions of the UK for the periods from 1996 to 2016.

```
head(hp)
```

```
##   Year North East North West Yorkshire and The Humber
## 1 1996      44500      46250                     46500
## 2 1997      46995      49000                     48950
## 3 1998      48000      50000                     50000
## 4 1999      50495      53500                     53000
## 5 2000      52000      56500                     56000
## 6 2001      55000      60000                     59950
##   East Midlands West Midlands  East London South East
## 1         47450         51000 58795  77000      68500
## 2         50000         54500 63500  86000      74500
## 3         53500         57000 69000  96500      83000
## 4         57500         60000 76000 118000      92500
## 5         61044         67000 86500 138000     112000
## 6         70000         75950 99995 155000     126000
##   South West Wales
## 1      57250 45000
## 2      60000 47950
## 3      66500 49500
## 4      74000 53000
## 5      85000 56500
## 6      96250 60000
```

The data is in wide format. If we want to plot a line for each region. we first need to set up a plotting region that is large enough. We use the common trick to plot `NULL`, which does not draw anything (we'll draw the lines later on).

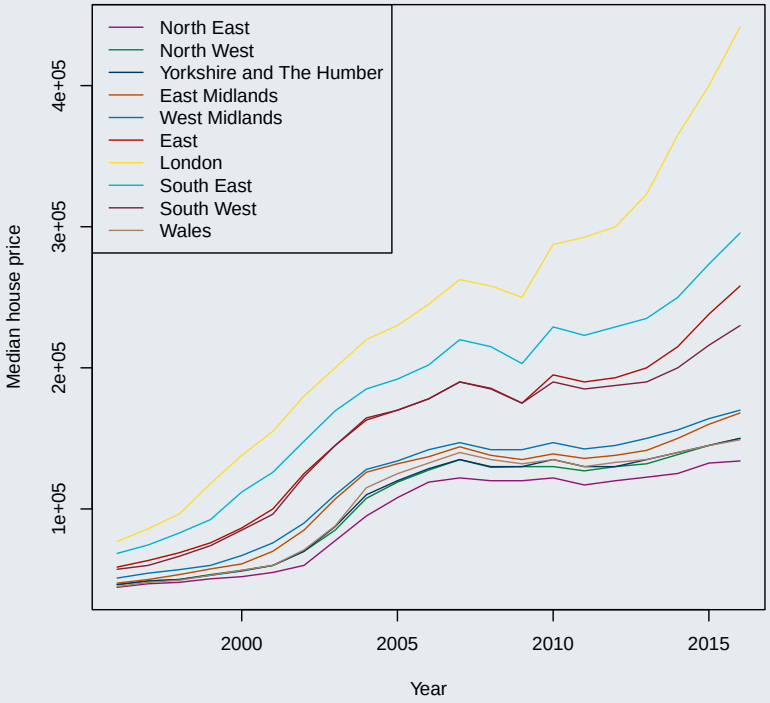
```
plot(NULL, xlim=range(hp$Year), ylim=range(hp[,-1]),
     xlab="Year", ylab="Median house price")
```

We now have an empty canvas, so we next add the lines corresponding to each region one by one. We will use a loop (we'll cover loops in more detail in a fortnight).

```
for (i in 2:ncol(hp))
    lines(hp$Year, hp[,i], col=i)
```

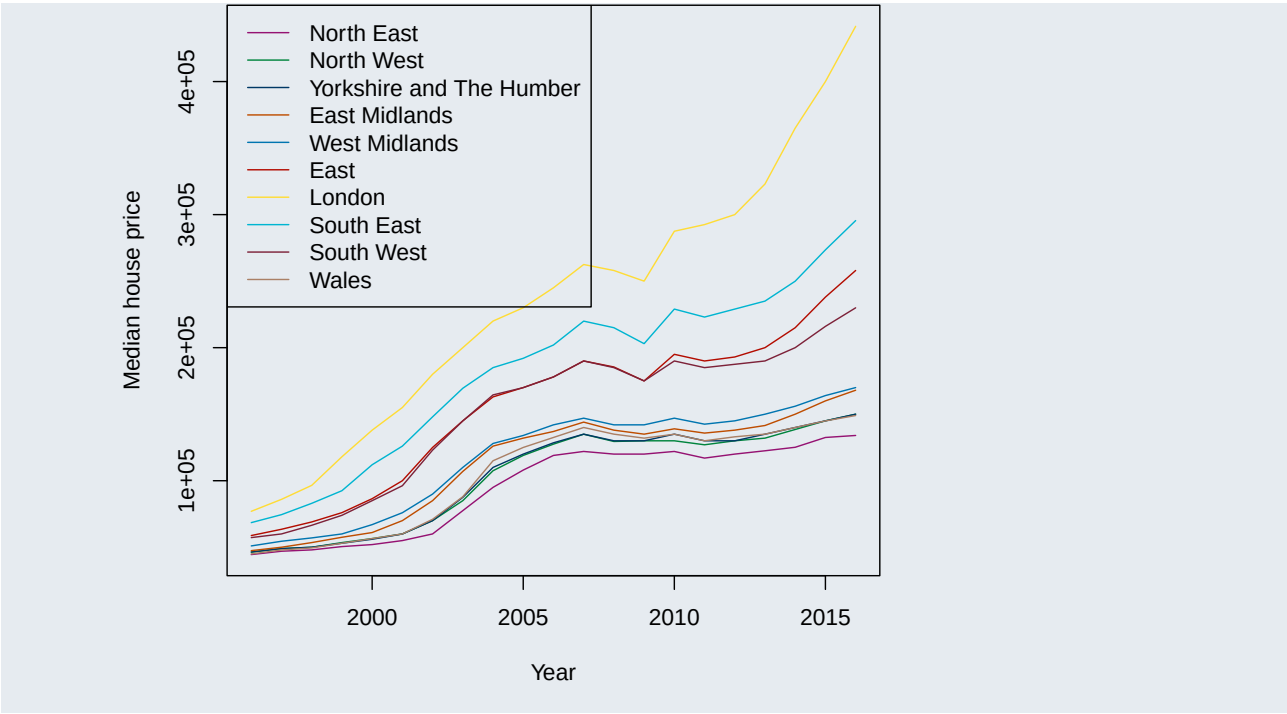
Finally we add a legend.

```
legend("topleft", lty=1, col=2:ncol(hp), colnames(hp)[-1])
```



This seemed rather complicated. Luckily enough, there is an R function `matplot`, which plots matrices column by column.

```
matplot(hp$Year, hp[,-1], type="l", col=2:ncol(hp), lty=1,
        xlab="Year", ylab="Median house price")
legend("topleft", lty=1, col=2:ncol(hp), colnames(hp)[-1])
```


North East
North West
Yorkshire and The Humber
East Midlands
West Midlands
East
London
South East
South West
Wales
Median house price
1e+05
2e+05
3e+05
4e+05
2000
2005
2010
2015
Year

Alternatives

ggplot2

The package ggplot2 is by far the most popular R package for graphics. It provides a declarative and simple, yet powerful interface for creating sophisticated graphics. We will look at ggplot2 in week 6.

lattice

The package lattice used to be popular before the advent of ggplot. It provides "Trellis"-like high-level plot functions (named after the plotting library for S-Plus) and has inspired some of the functionality of ggplot.

3D plots using rgl

The package rgl allows for creating more sophisticated 3D plots using OpenGL. OpenGL is a cross-platform 3D graphics rendering engine like DirectX under Windows. OpenGL is highly sophisticated: it supports advanced features like translucent objects, textured surfaces, light effects and even reflective surfaces. The package rgl allows rendering 3D plots through OpenGL. `rgl` plots can be spun using your mouse.

Example 19.

We can create a 3D `rgl` scatter plot of the health expenditure data using

```
library(rgl)
plot3d(health[,4:6], type="s", radius=1e7, col=unclass(health$Region), alpha=0.5)
```

Javascript libraries

If you want to publish your plots on the web it is worth considering JavaScript-based plotting libraries such as plotly or D3. Plots generated from R are static images whereas these libraries allow for user interaction.

The package plotly allows creating plotly graphics from R. These graphics allow interactive features such as selecting variables, zooming in plots and subsetting sections of plots.

Plotly documentation for R

https://plot.ly/r/

The documentation for the R interface to plotly contains a large number of examples.

Integrating R and D3 is slightly more complicated, but can be done using the R package R2D3. It however still requires to to construct the D3 plot in Javascript. It is possible to use d3 plots in Shiny (we will look at Shiny later on in this course).

R2D3 documentation

https://rstudio.github.io/r2d3/

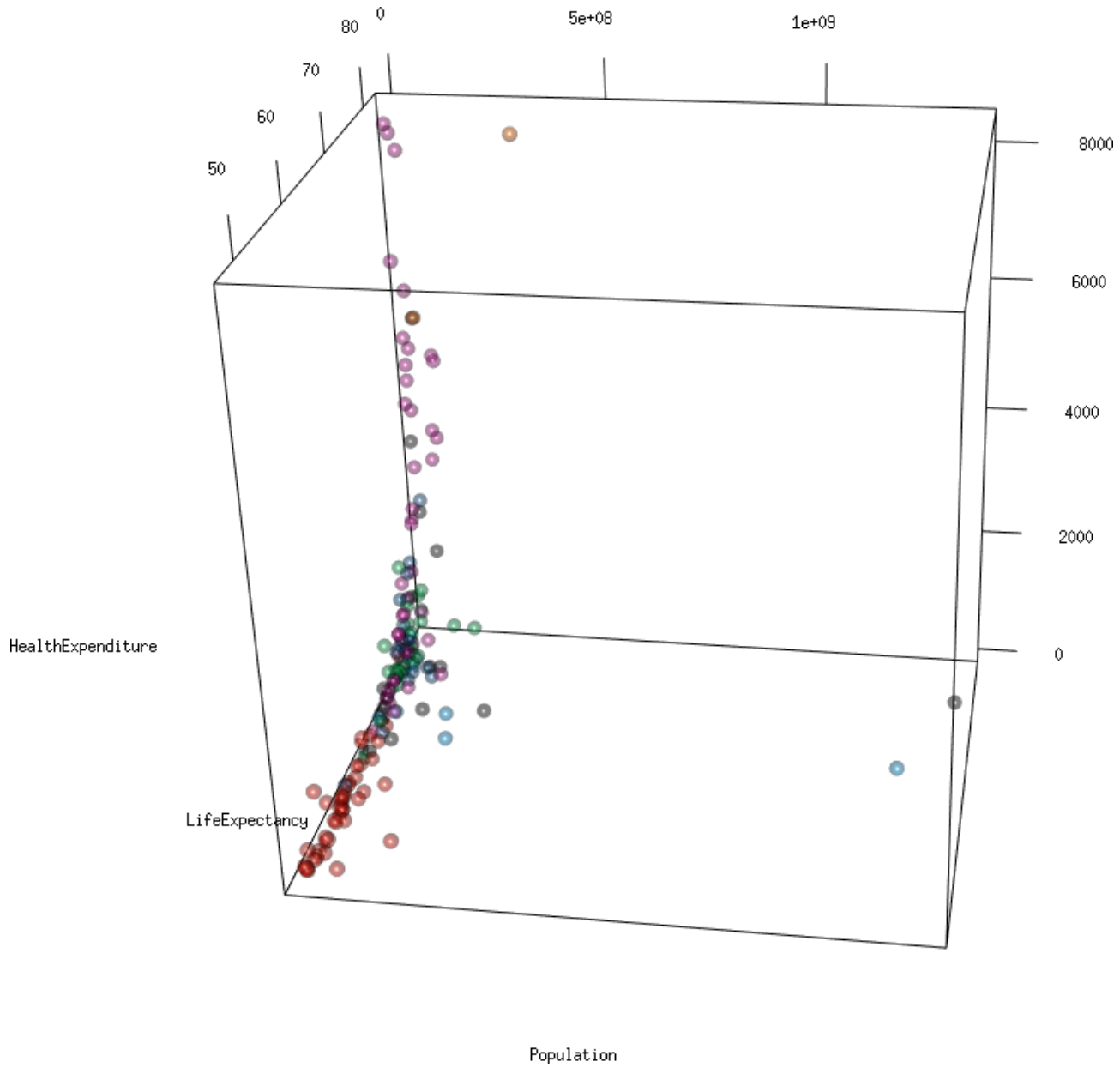
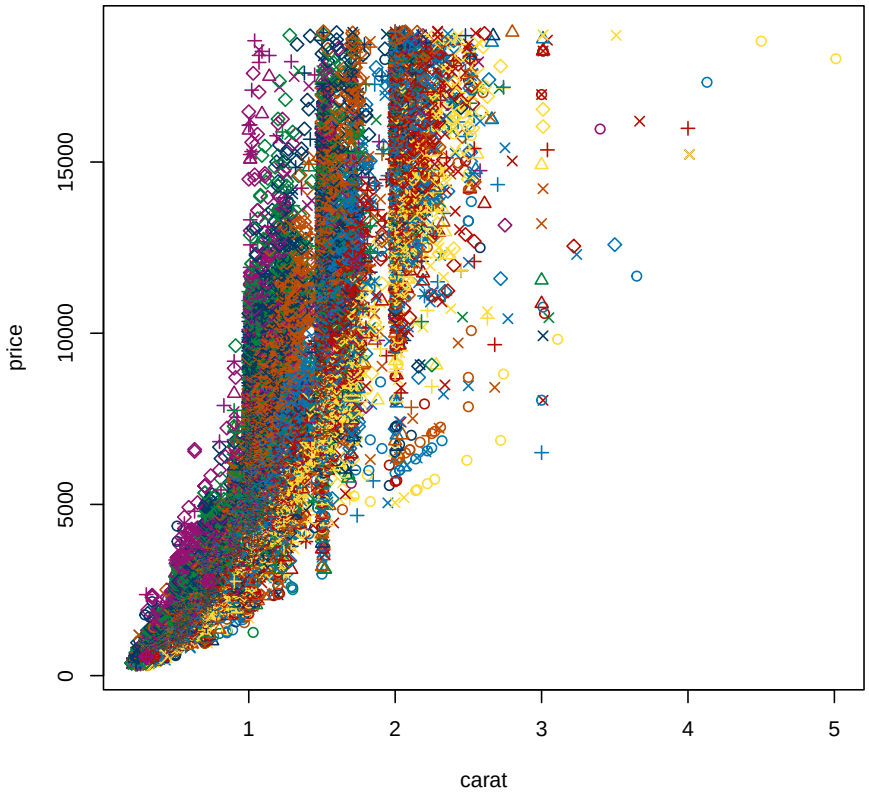


Figure 4: Screenshot of the `rgl` plot from the example

Answers to tasks

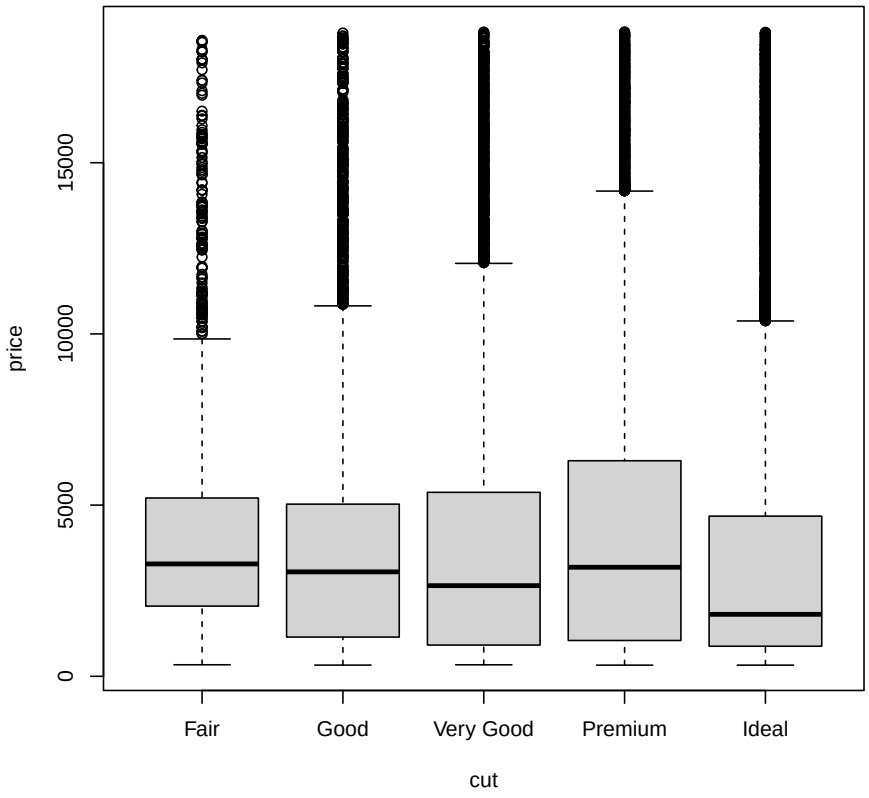
Answer to Task 1. We should also add a legend, which we will soon learn about.

```
plot(price~carat, data=diamonds, col=unclass(color)+1, pch=unclass(cut))
```

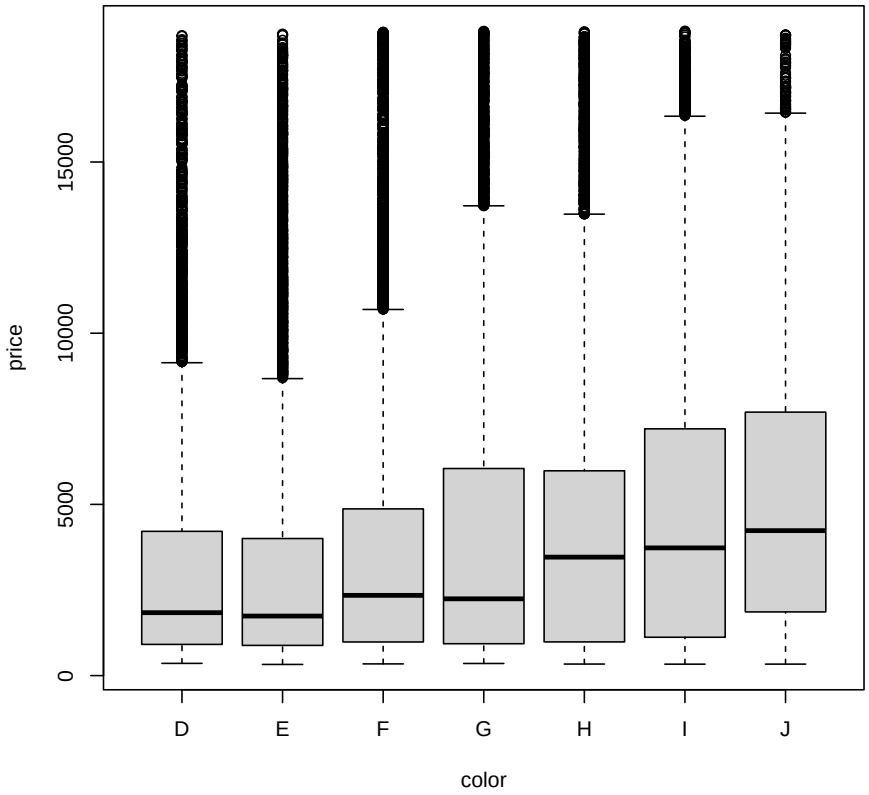


Answer to Task 2. We can use the following R code:

```
library(tidyverse)
diamonds %$%
   boxplot(price ~ cut)
```

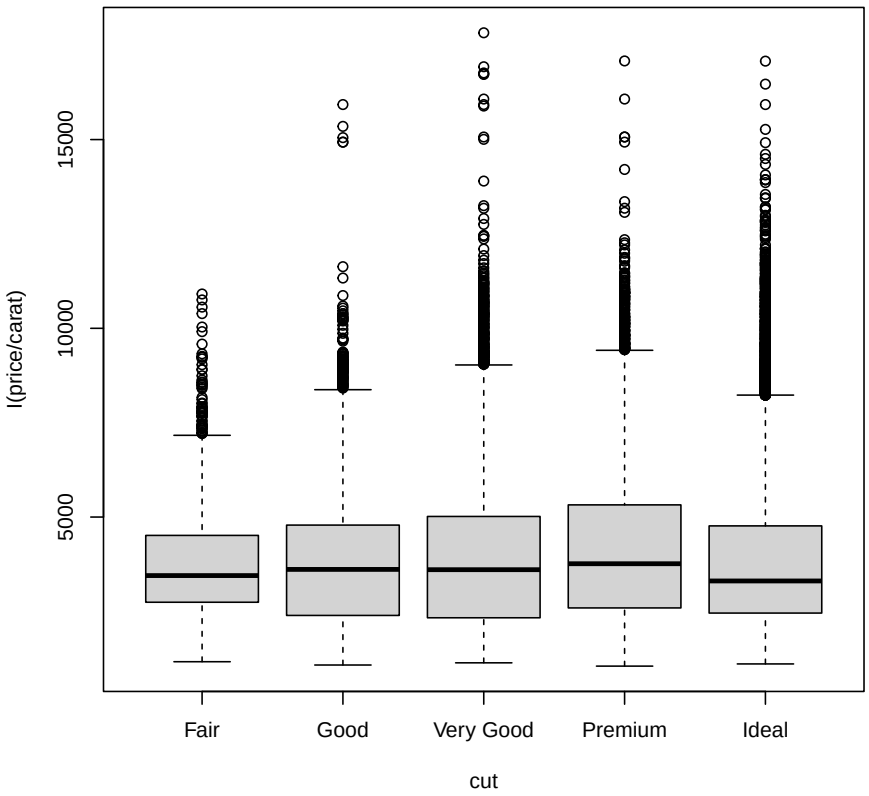


```
diamonds %$%
   boxplot(price ~ color)
```



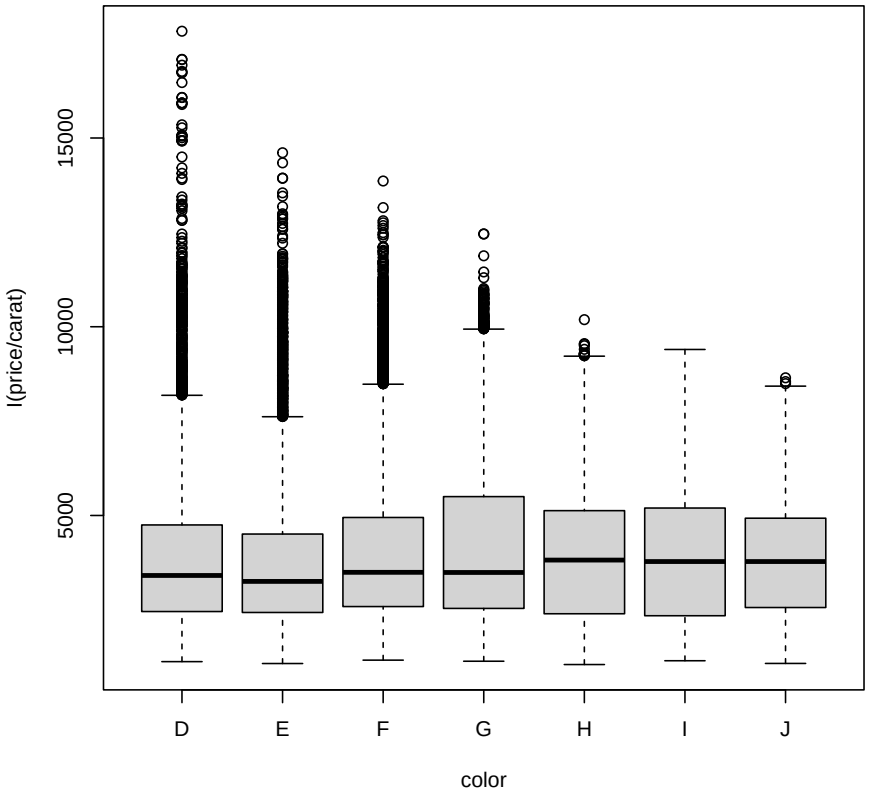
This seems odd: the better the diamond (lower letters denote better colours), the lower the price. The reason for this is that we forgot to take the size of the diamond into account. Perfect diamonds tend to be small. The story changes a little if we look at the price per weight.

```
diamonds %$%
   boxplot(I(price/carat) ~ cut)
```



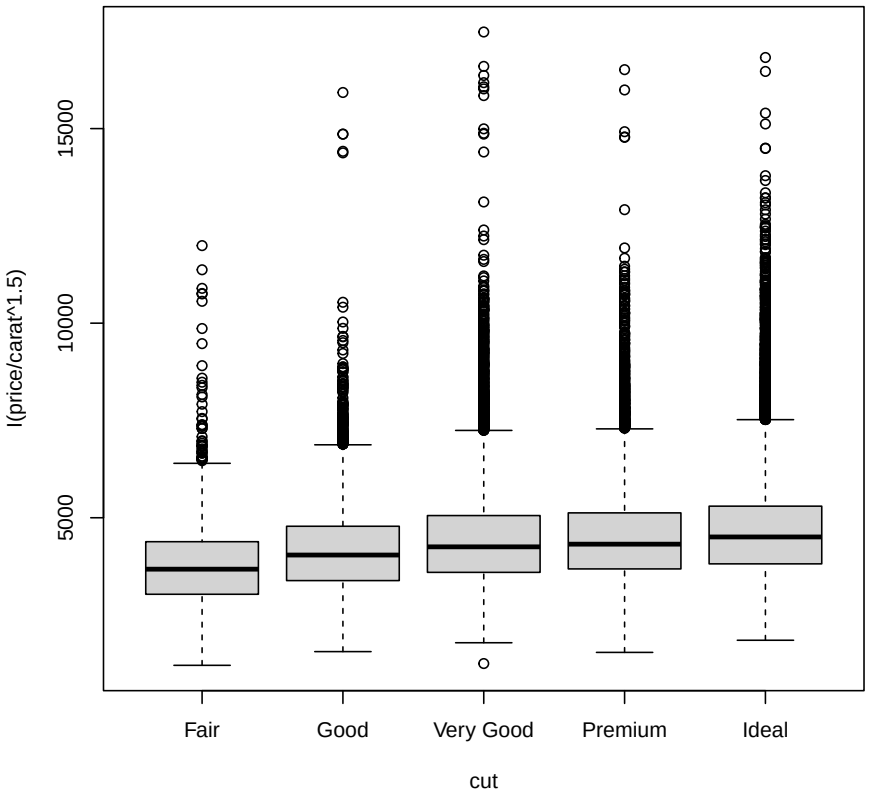

```
diamonds %$%
```

```
boxplot(I(price/carat) ~ color)
```

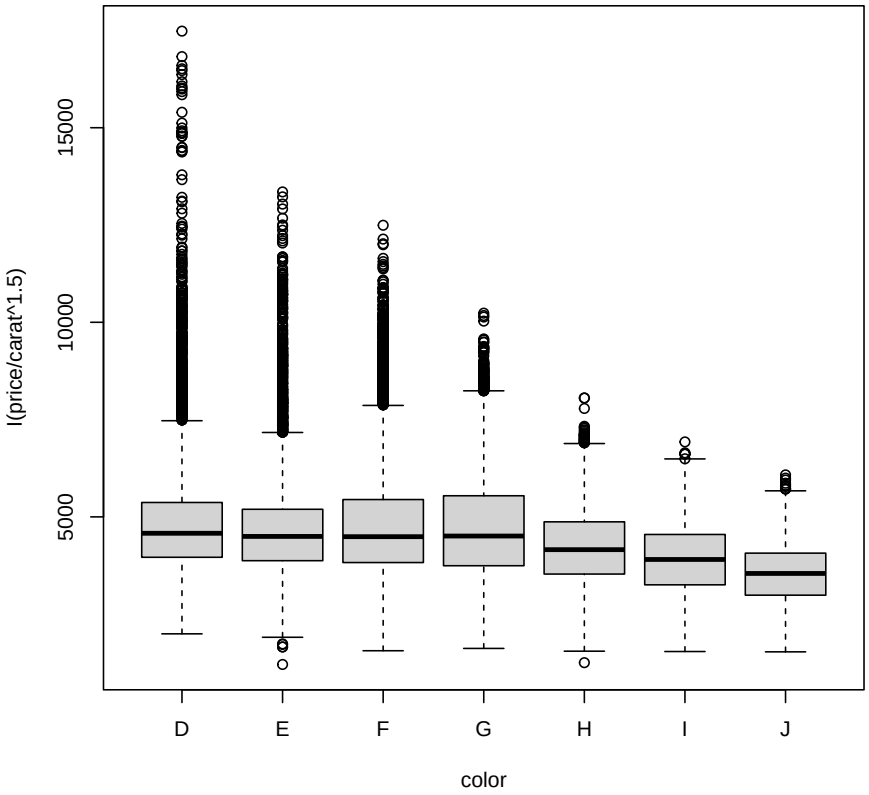


The problem is in some way that because large diamonds are rare, they are disproportionately expensive. If we take carat to the power of 1.5 we get

```
diamonds %$%
   boxplot(I(price/carat^1.5) ~ cut)
```




```
diamonds %$%
   boxplot(I(price/carat^1.5) ~ color)
```



which looks more plausible.

The function `I` lets us perform transformations inside the plotting functions.

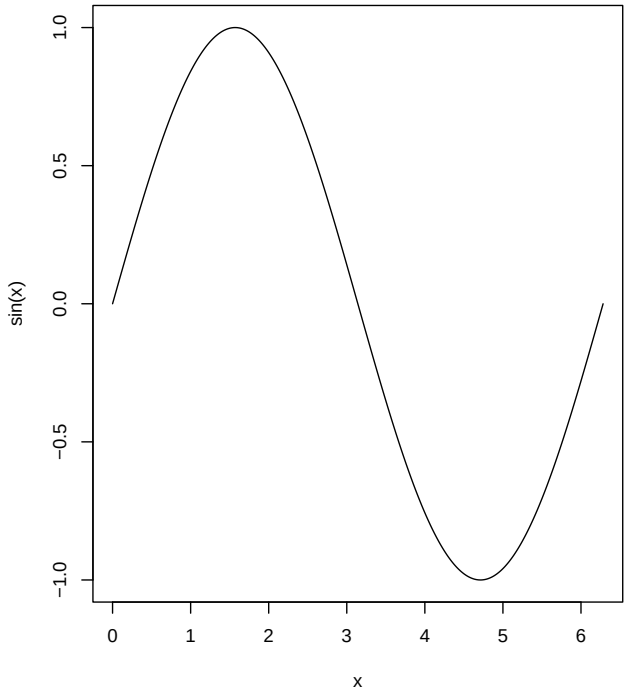
Answer to Task 3. The former set of commands first draws the noisy data and then adds the sine curve. The latter set of commands first draws the sine curve and then the noisy data. The noisy data have a greater range than the noise-free data. As `plot` determines the range of the y axis and this cannot be changed by `points` or `lines`,drawing the noisy points first (as done by the former command) gives a larger range of the y axis that can show all the data. Drawing the noise-free line first leads to a smaller range of the y axis, which is then too small to show all the noisy points.

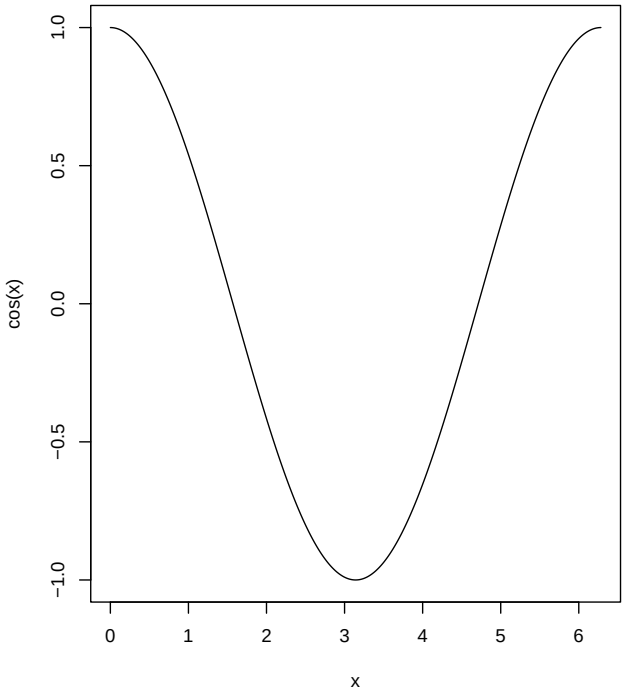
Thus the left-hand plot comes from the latter command (noise-free line first, then noisy points) and the right-hand plot comes from the former command (noisy points first, then noise-free line).

Another difference is that the former command plots the line above the points, whereas the latter plots the points above the line.

Answer to Task 4. In this example it does not matter whether we use `mrfow` or `mfcol` as there are only two plots (so it doesn't matter whether we go column-wise or row-wise).

```
x <- seq(0, 2*pi, length.out=1000)
par(mfrow=c(1, 2))
plot(x, sin(x), type="l")
plot(x, cos(x), type="l")
```





The function `curve` makes sketching curves easier.

```
par(mfrow=c(1, 2))
curve(sin, from=0, to=2*pi)
curve(cos, from=0, to=2*pi)
```