STAT 210
Applied Statistics and Data Analysis:
Graphics in R

Joaquín Ortega

Fall, 2020

Contents

1 Graphics in R

One of the main features of `R` is the variety of graphical tools that are included as part of the base package.

Main graphical packages in `R`:

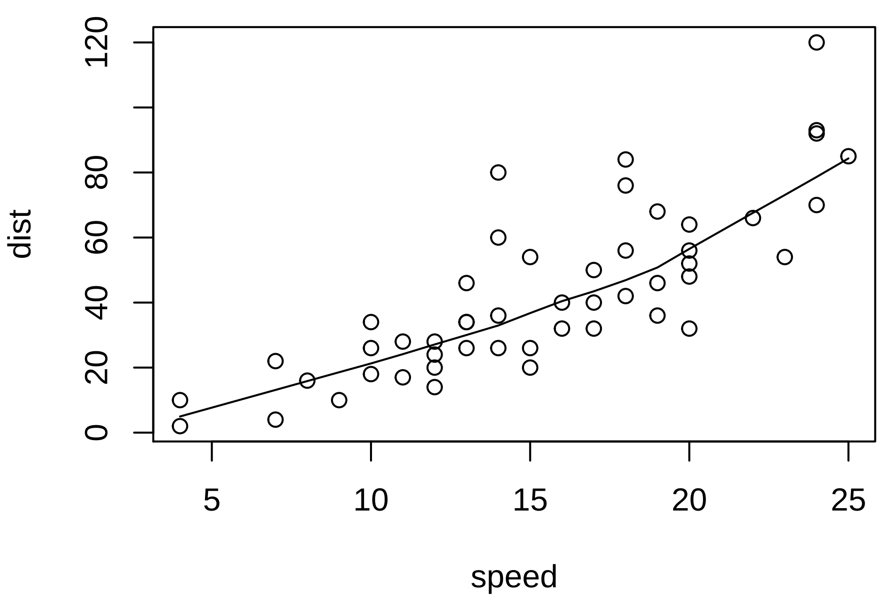
- `base`
- `lattice` by Deepayan Sarkar
- `ggplot2` by Hadley Wickham
- interactive packages such as `shiny` and `plotly`.

We will only consider here the `base` package (although some of the plots in the previous presentation were made with `lattice` and `ggplot2`).

We start with a demo of the graphical capabilities of `R` using the functions `example` and `demo`. We will run only one of these functions. You should run the other examples in your computer

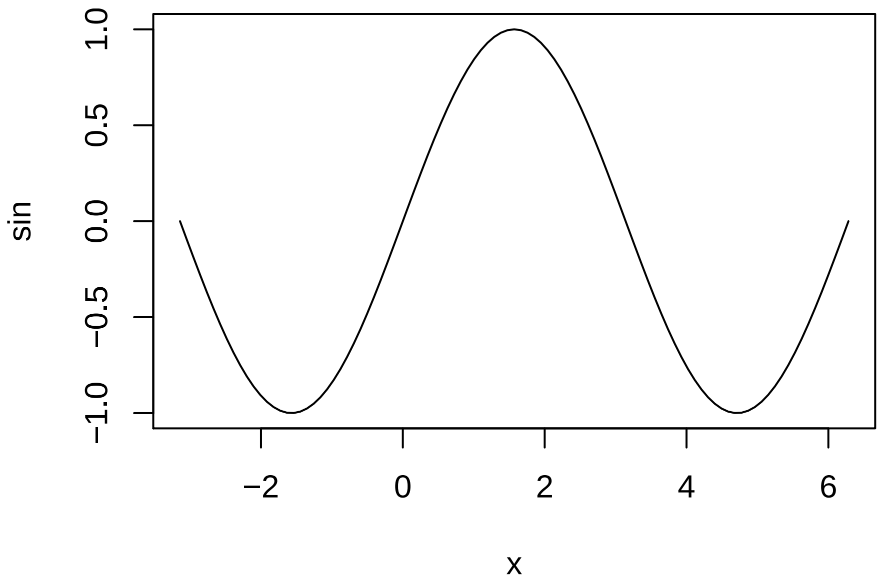
```
example(plot)
```

```
##
## plot> require(stats) # for lowess, rpois, rnorm
##
## plot> plot(cars)
```

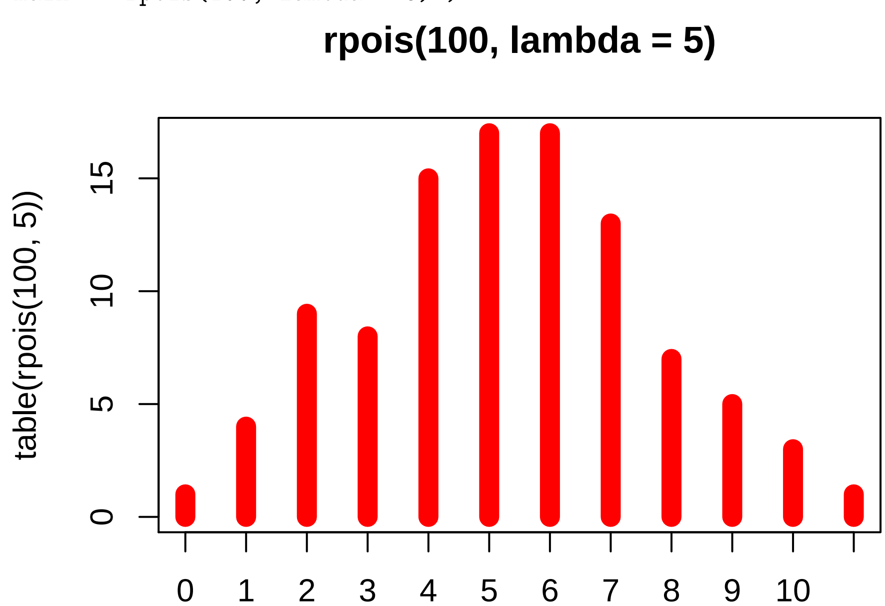



```
##
## plot> lines(lowess(cars))
##
```

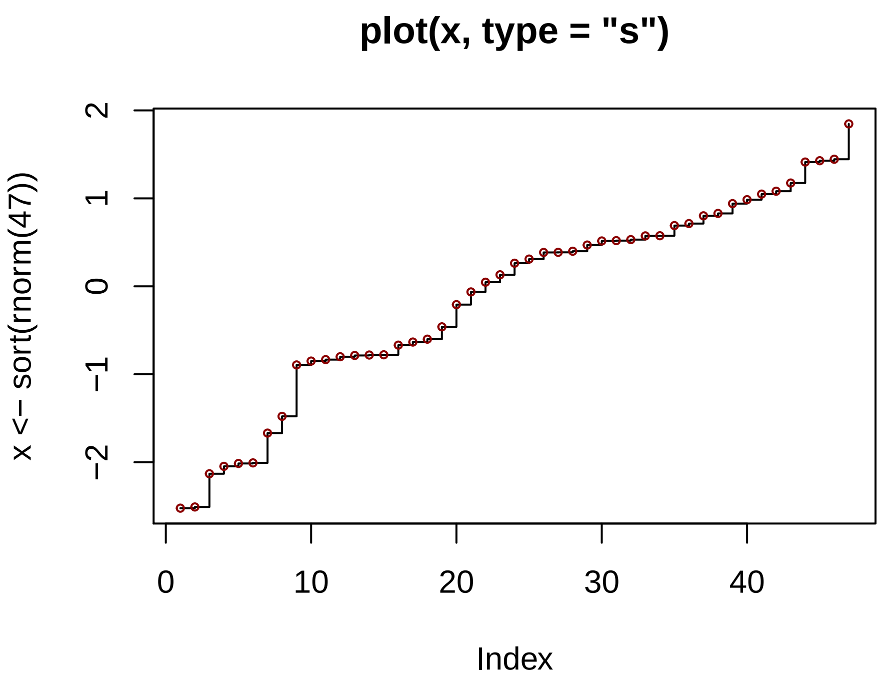
```
## plot> plot(sin, -pi, 2*pi) # see ?plot.function
```




```
##
## plot> ## Discrete Distribution Plot:
## plot> plot(table(rpois(100, 5)), type = "h", col = "red", lwd = 10,
## plot+      main = "rpois(100, lambda = 5)")
```




```
##
## plot> ## Simple quantiles/ECDF, see ecdf() {library(stats)} for a better one:
## plot> plot(x <- sort(rnorm(47)), type = "s", main = "plot(x, type = \"s\")")
```




```
##
## plot> points(x, cex = .5, col = "dark red")
```

The `example` function runs the commands on the 'Examples' section of the help file. In the previous example, four plots were drawn, and the commands for these plots can be found in the Examples section of `help(plot)`. The function `example` can be used with any `R` function, as long as the help file for the function has examples. Here are two for you to run:

```
example(par)
example("palette")
```

The function `demo`, on the other hand, runs demonstrations on certain topics. To see the list of available topics type `demo()` and a window will open in RStudio with the list of available topics. You should run the following:

```
demo(graphics)
demo(persp)
demo(image)
demo(colors)
```

2 High level plotting commands

High level plotting commands produce graphs. The main high level command is `plot`, but there are others such as`boxplot`, `barplot`, `histogram` and `dotchart`. We will review some of them in what follows. Low level commands add additional graphical elements or text to a graph.

3 `plot`

`plot()` is the standard function for plotting in `R`. The output depends on the object that holds the data, the mode of the data and the syntax you use. Let's see some examples using the data set `iris`.

```
attach(iris)
plot(Sepal.Length, Sepal.Width)
```

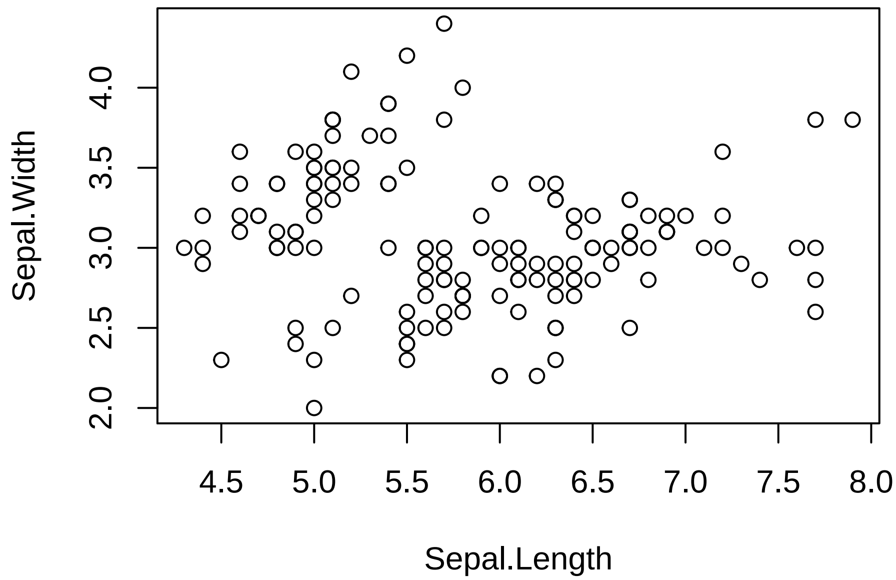


Figure 1: Plot of ‘Sepal.Width‘ against ‘Sepal.Length‘ for the ‘iris‘ dataset using the ‘plot‘ function

The same result is obtained with the following two commands

```
plot( Sepal.Width ~ Sepal.Length)
plot(~ Sepal.Length + Sepal.Width)
```

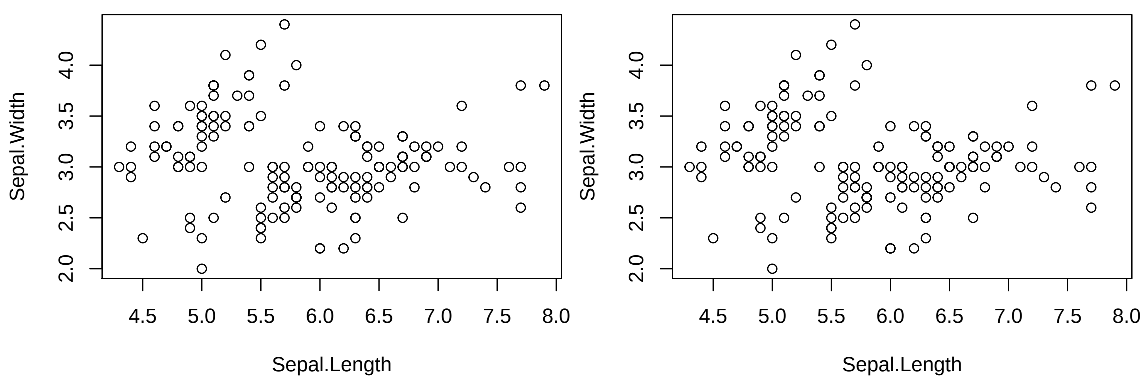


Figure 2: Plot of ‘Sepal.Width‘ against ‘Sepal.Length‘ for the ‘iris‘ dataset using the ‘plot‘ function

If the argument to plot is a factor, then `plot` produces a bar graph

```
plot(Species)
```

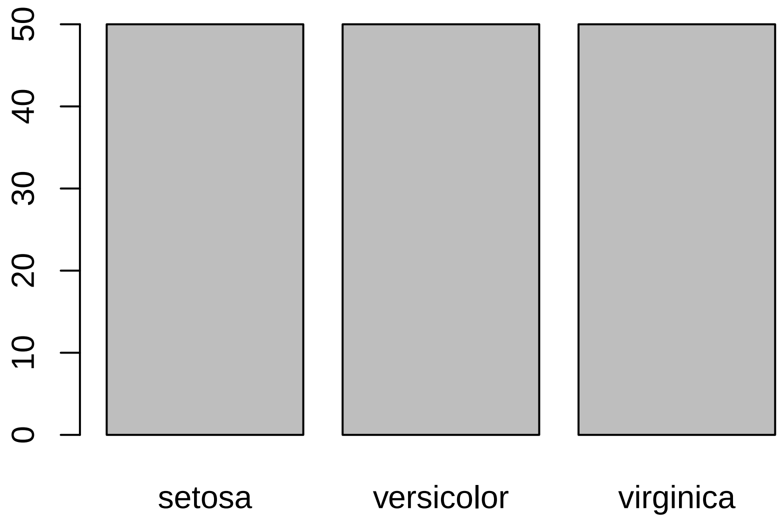


Figure 3: Bar graphs for the number of data points corresponding to the three species in the ‘iris‘ dataset.

In this case the graph is not very interesting since the data set has the same number of cases for each of the three species considered.

If a factor (i.e., a categorical variable) goes to the x-axis, the result is a boxplot:

```
plot(Species, Sepal.Length)
```

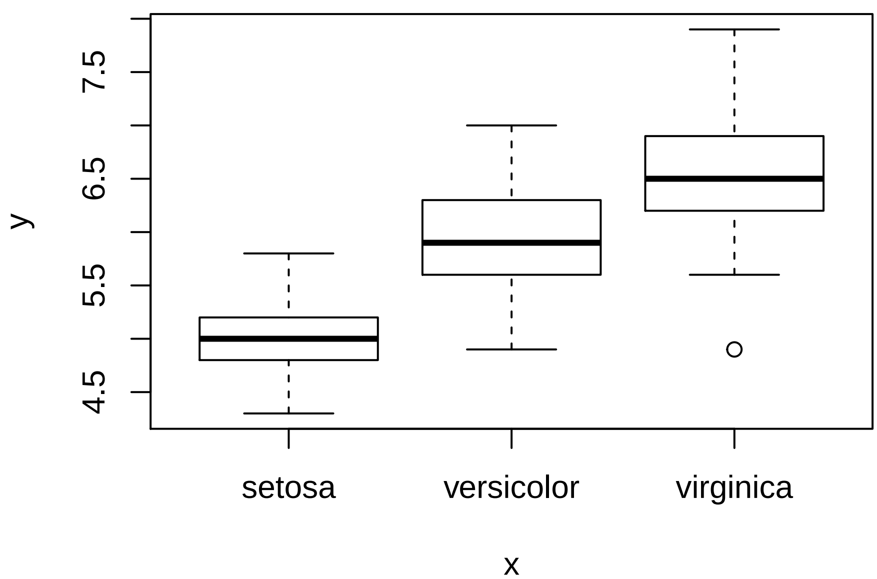


Figure 4: Boxplots for Sepal Length corresponding to the three species in the ‘iris‘ dataset.

and if the argument is the whole data set, we get a matrix of graphs, with all possible combinations of pairs of variables. Observe that the values for `Species`, which is a factor, are represented as integers.

```
plot(iris)
```

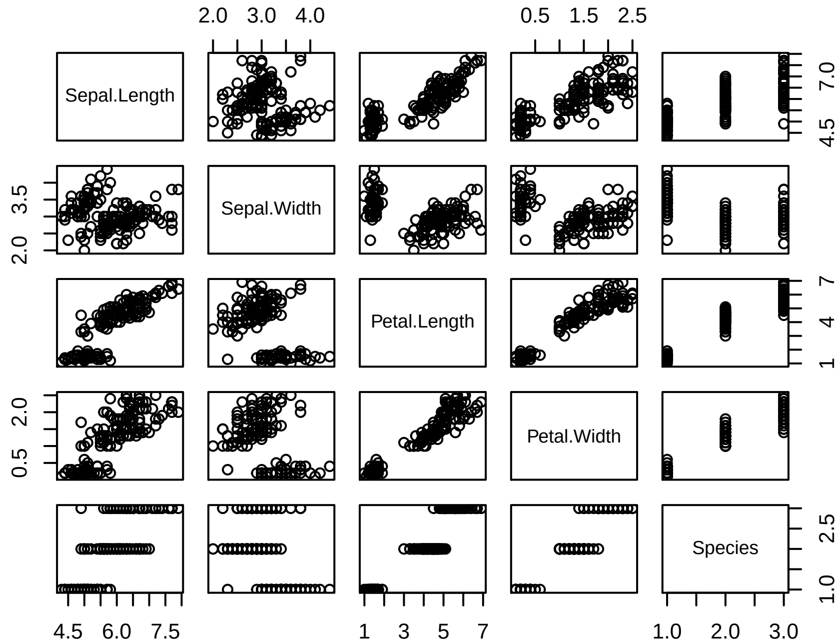


Figure 5: Paired graphs for the variables in the 'iris' dataset.

If we use a formula with two variables on the right-hand side, we get two graphs with the same variable on the y-axis:

```
par(mfrow=c(1,2))
plot(Petal.Length ~ Sepal.Width + Sepal.Length)
```

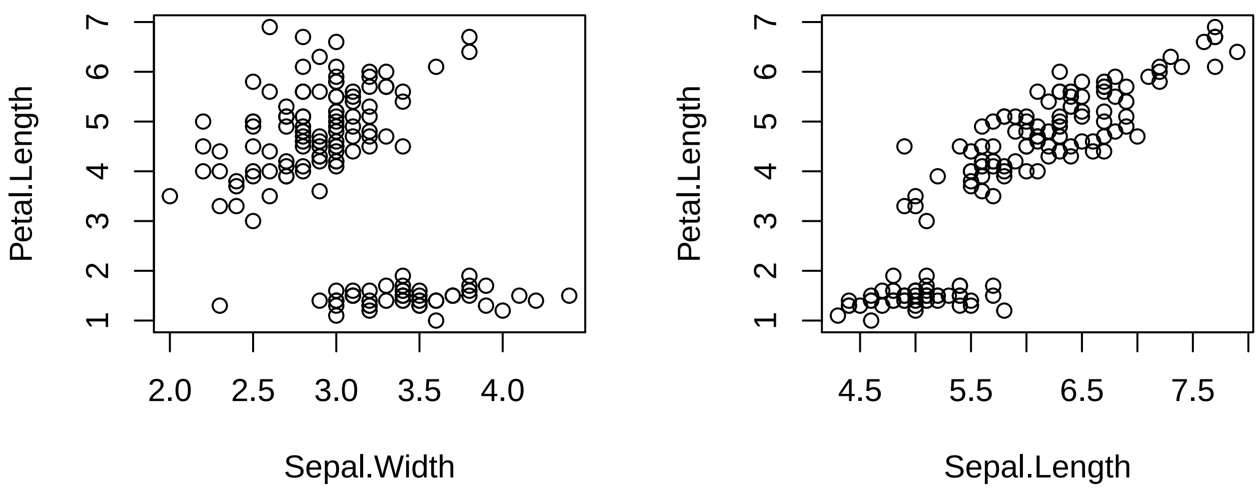


Figure 6: 'Petal.Length' as a function of 'Sepal.Width' and 'Sepal.Length' for the 'iris' dataset

3.1 Options

After the main argument that specifies the variables to be plotted, the function `plot` accepts options that modify the plot.

3.1.1 `type`

The `type` option determines the type of plot to be produced. The options are listed in table 2.1.

Option	Value
`type = 'p'`	Plots points, is the default option
`type = 'l'`	Plots lines.
`type = 'b'`	Plots points joined by lines.
`type = 'o'`	Points and lines are superimposed.
`type = 'h'`	Vertical lines.
`type = 's'`	Step function, continuous from right.
`type = 'S'`	Step function, continuous from left.
`type = 'n'`	Does not draw the graph but keeps the dimensions

Table 2.1 Options for the 'plot' function.

To show the effect that these options have on the plot we will use the `cars` data set, that has the speed and stopping distance for 50 cars, measured in the 1920's.

```
str(pressure)
```

```
## 'data.frame':    19 obs. of  2 variables:
##  $ temperature: num  0 20 40 60 80 100 120 140 160 180 ...
##  $ pressure   : num  0.0002 0.0012 0.006 0.03 0.09 0.27 0.75 1.85 4.2 8.8 ...
```

The option `type = 'p'` plots points.

```
plot(pressure, type='p')
```

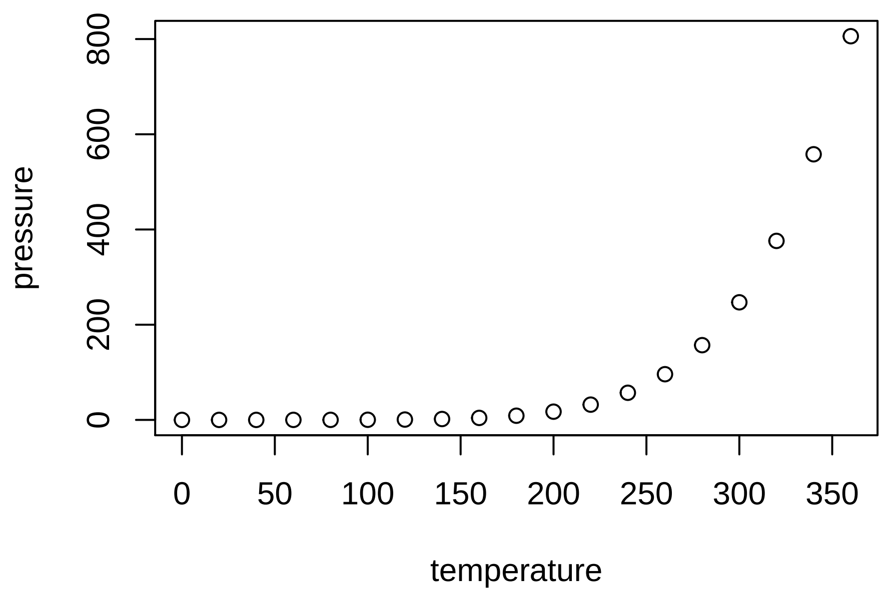


Figure 7: Graph of pressure against temperature with the option 'type = 'p''

`type = 'l'` plots a line joining the consecutive points:

```
plot(pressure, type='l')
```

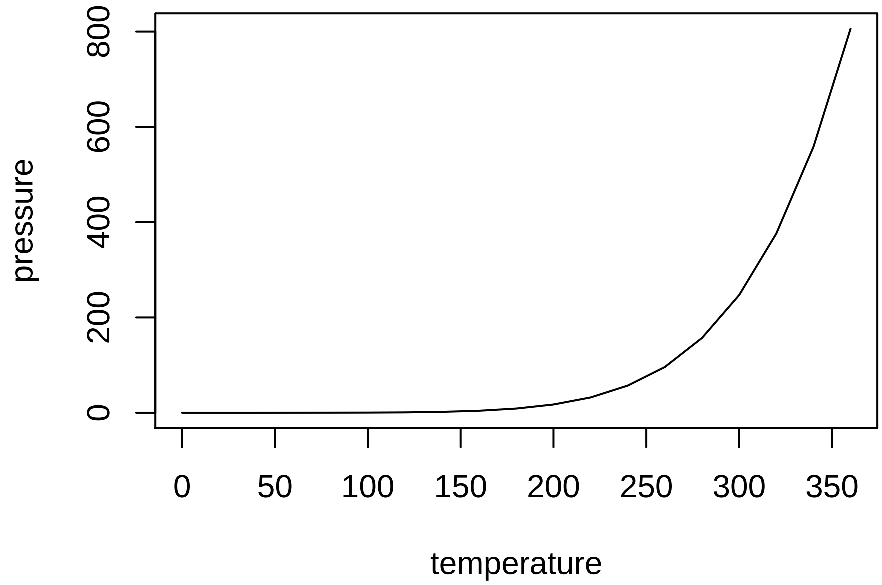


Figure 8: Graph of pressure against temperature with the option 'type = 'l''

`type = 'b'` plots both lines and points:

```
plot(pressure, type='b')
```

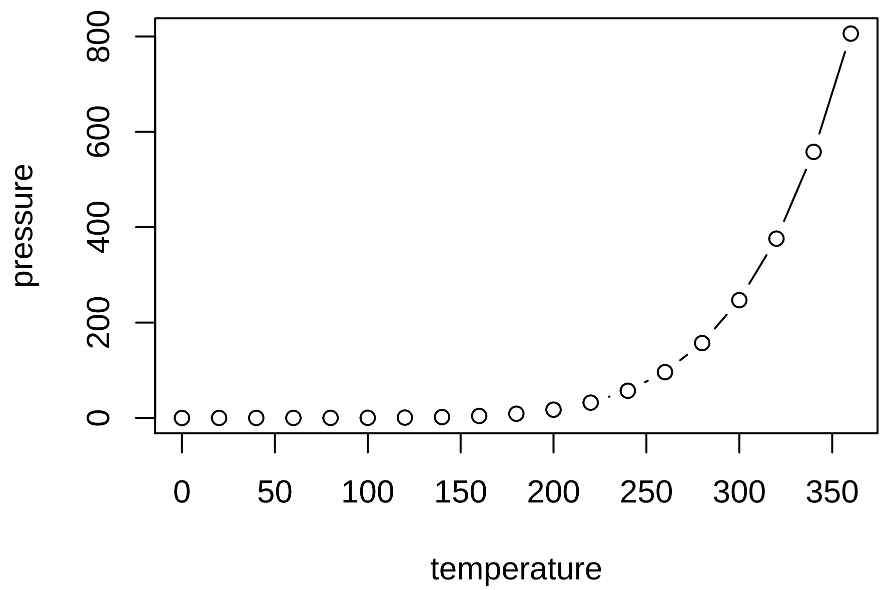


Figure 9: Graph of pressure against temperature with the option 'type = 'b''

`type = 'o'` also plots both lines and points, but now the lines overplot the points

```r
plot(pressure, type='o')
```

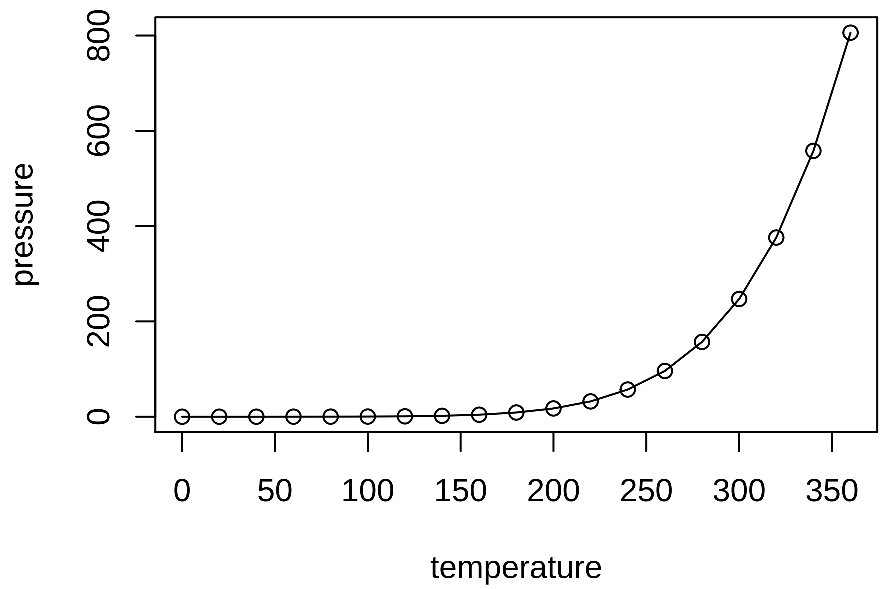


Figure 10: Graph of pressure against temperature with the option ‘type = ’o’‘

`type = 'h'` is a histogram-like plot, with vertical lines

```r
plot(pressure, type='h')
```

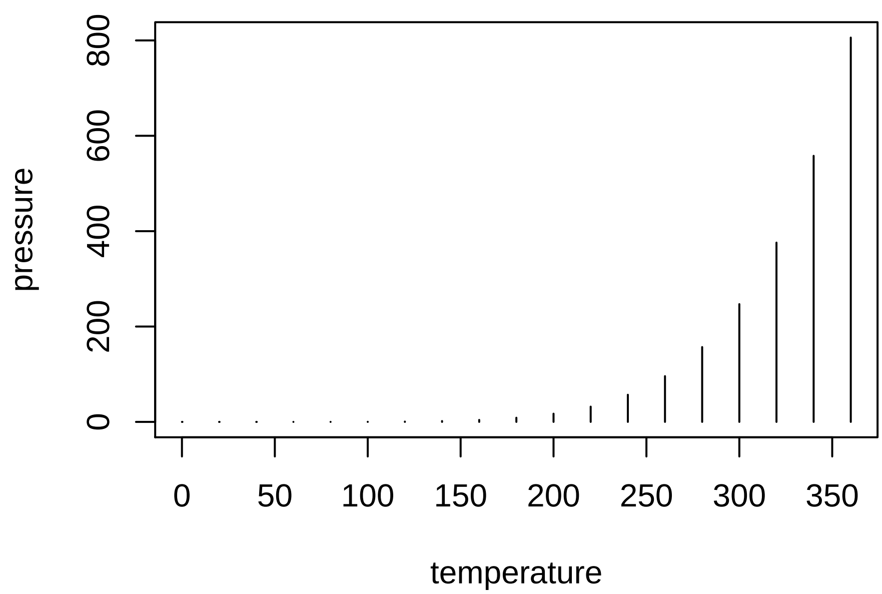


Figure 11: Graph of pressure against temperature with the option ‘type = ’h’‘

`type = 'n'` does not plot the graphs but plots the bounding box and the scales on the axes.

```r
plot(pressure, type='n')
```

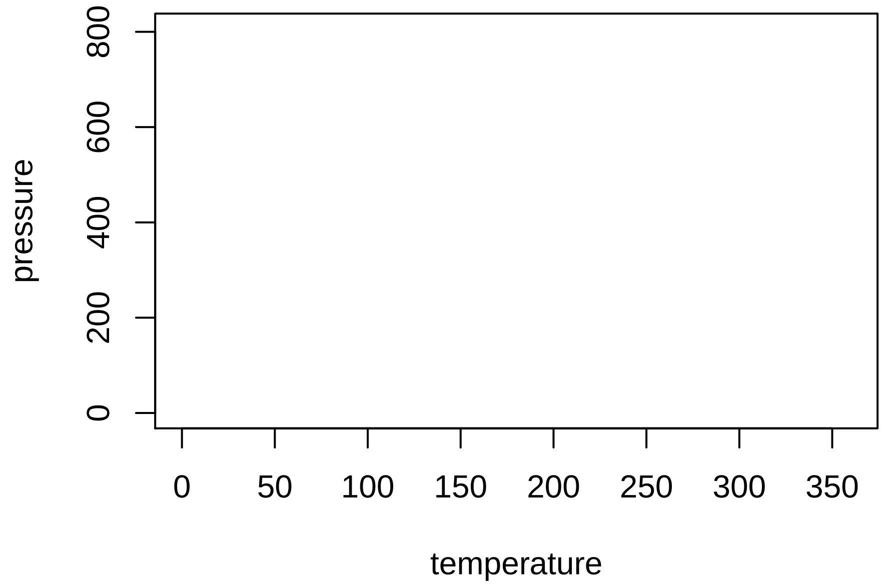


Figure 12: Graph of pressure against temperature with the option ‘type = ’n’‘

Finally, `'s'` and `'S'` plot a stair function but the points are at the origin of the horizontal segments for `'s'` and at the end for `'S'`.

```
par(mfrow=c(1,2))
plot(pressure,type='s')
points(pressure)
plot(pressure,type='S')
points(pressure)
```

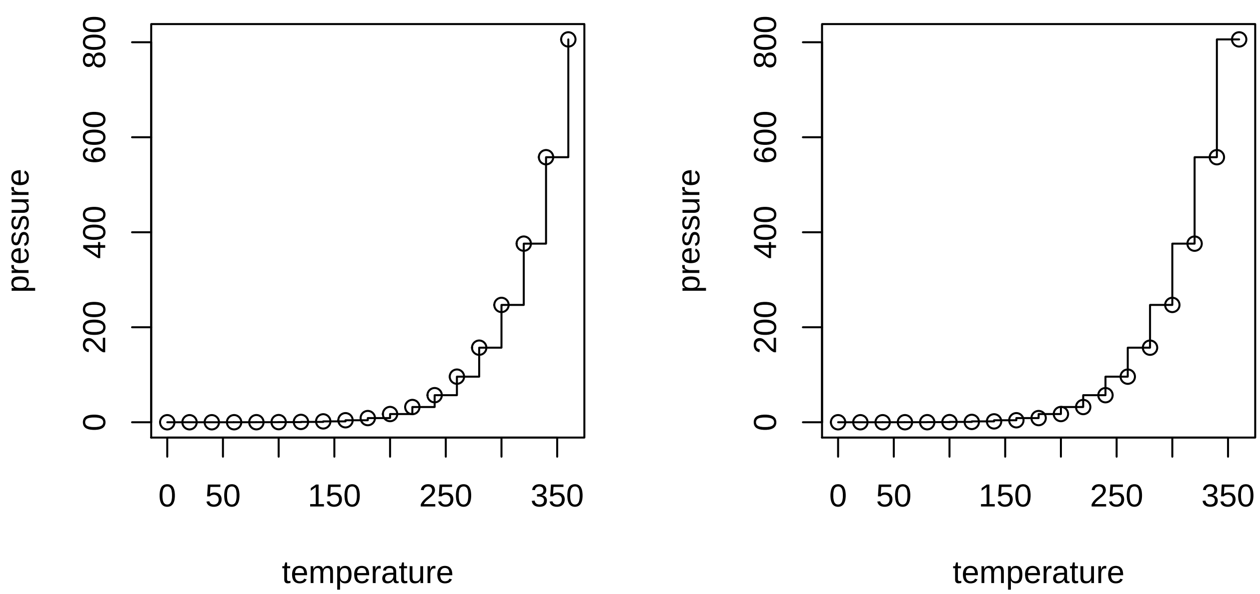


Figure 13: Graph of pressure against temperature with the option ‘type = ’s’‘ (left) and ‘type = ’S’‘ (right)

```
par(mfrow=c(1,1))
```

3.1.2 `xlab` and `ylab`

The options `xlab` and `ylab` give customized labels for the axes. By default, the labels come from the names of the variables in the data object.

```
plot(pressure, xlab = 'Temperature (ºC)', ylab = 'Pressure (mm of Hg)')
```

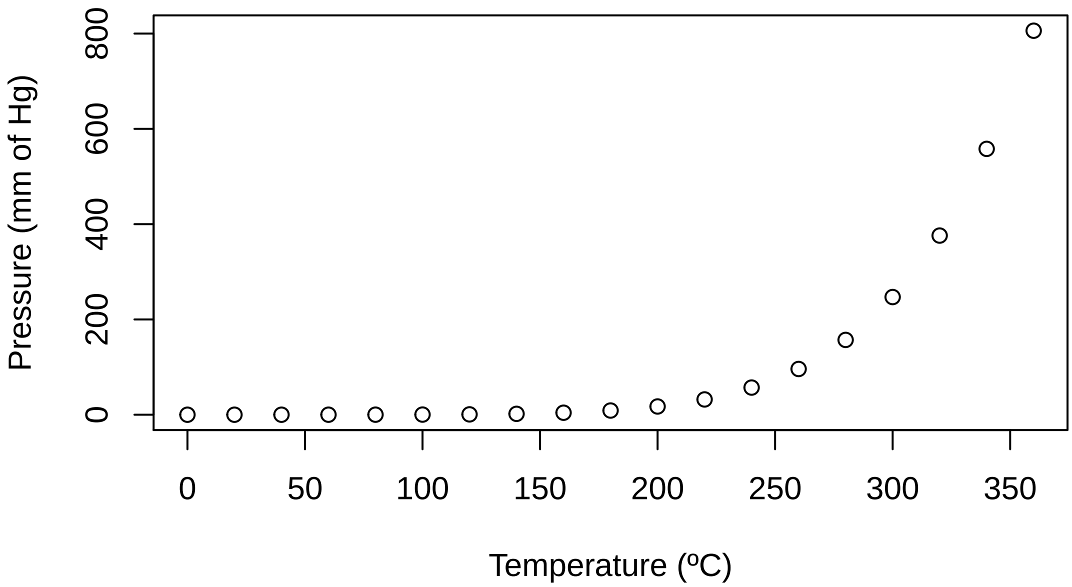


Figure 14: Graph of pressure against temperature with customized axes labels

3.1.3 `main` and `sub`

These options give a title and subtitle to the graph.

```
plot(pressure,type='l', xlab = 'Temperature (ºC)', ylab = 'Pressure (mm of Hg)',
     main='Pressure data',sub='Vapor pressure of mercury')
```

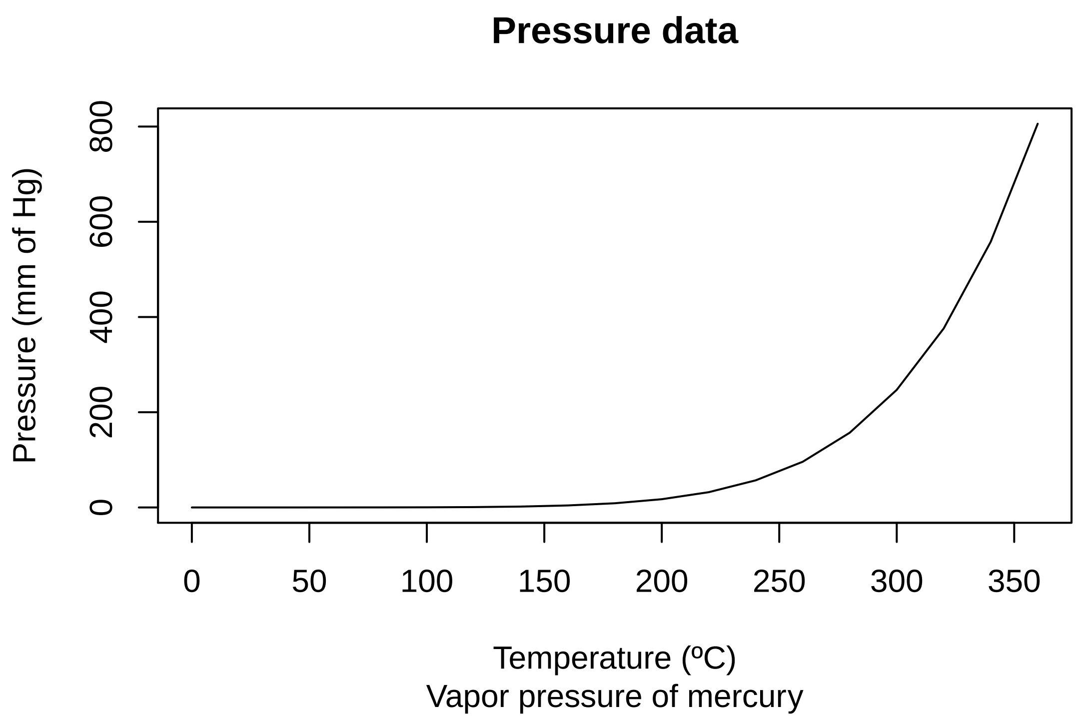


Figure 15: Graph of pressure against temperature with the option 'type = 'l'' and title and subtitle

3.1.4 `asp`

The option `asp` controls the y/x aspect ratio. One data unit in the x direction equals in length `asp`*1 data units in the y direction.

```
par(mfrow = c(1,2))
plot(pressure,type='l', xlab = 'Temperature (ºC)', ylab = 'Pressure (mm of Hg)',
     main='Pressure data', sub='Vapor pressure of mercury', asp=1)
plot(pressure,type='l', xlab = 'Temperature (ºC)', ylab = 'Pressure (mm of Hg)',
     main='Pressure data',sub='Vapor pressure of mercury', asp=0.1)
```

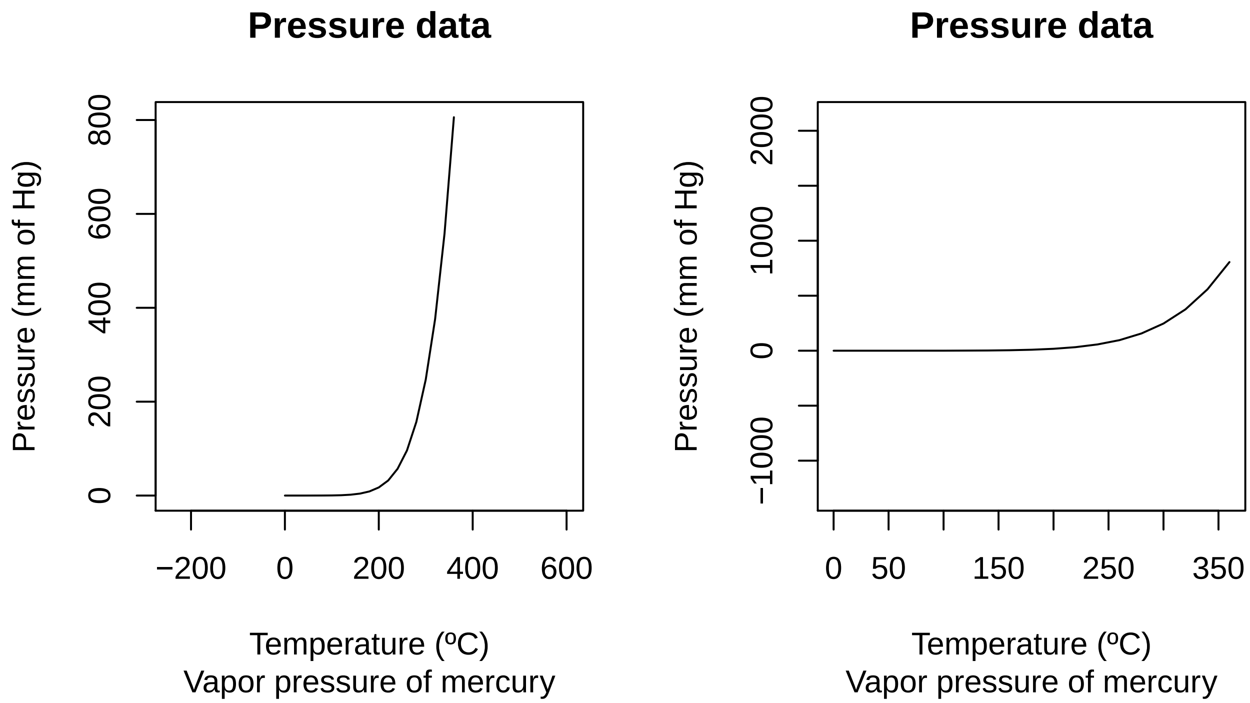


Figure 16: Graph of pressure against temperature with the option ‘type = ’l"‘ and two different values for the option ‘asp‘

3.1.5 `lty` and `lwd`

These two options control the type of line and the width, respectively, for the lines in the plot. There are six standard types of line. The following function from the help for the `par` function draws all standard line types in two different width.

```
showLty <- function(ltys, xoff = 0, ...) {
   stopifnot((n <- length(ltys)) >= 1)
   op <- par(mar = rep(.5,4)); on.exit(par(op))
   plot(0:1, 0:1, type = "n", axes = FALSE, ann = FALSE)
   y <- (n:1)/(n+1)
   clty <- as.character(ltys)
   mytext <- function(x, y, txt)
      text(x, y, txt, adj = c(0, -.3), cex = 0.8, ...)
   abline(h = y, lty = ltys, ...); mytext(xoff, y, clty)
   y <- y - 1/(3*(n+1))
   abline(h = y, lty = ltys, lwd = 2, ...)
   mytext(1/8+xoff, y, paste(clty," lwd = 2"))
}
showLty(c("solid", "dashed", "dotted", "dotdash", "longdash", "twodash"))
```

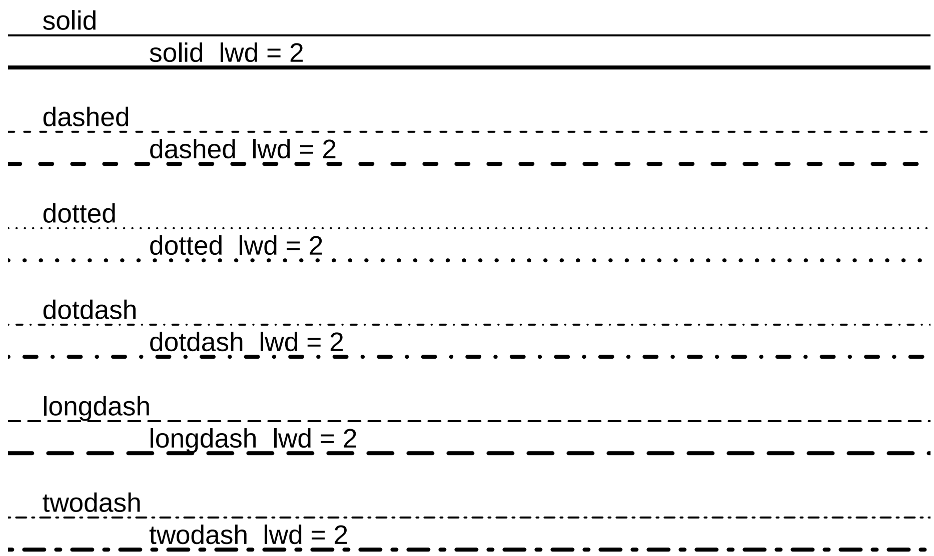


Figure 17: The six different line types, each in two different widths

Additionally, line types can be specified directly by a sequence of numbers that indicate the length of on/off segments of line. For this, a sequence with an even number of components (up to eight) is required, each component is a non-zero digit. For example, `3212` specifies three units on followed by two units off followed by one on and finally two off.

```
plot(pressure,type='l', xlab = 'Temperature (ºC)', ylab = 'Pressure (mm of Hg)',
     main='Pressure data', sub='Vapor pressure of mercury', lty = '3214', lwd = 2)
```

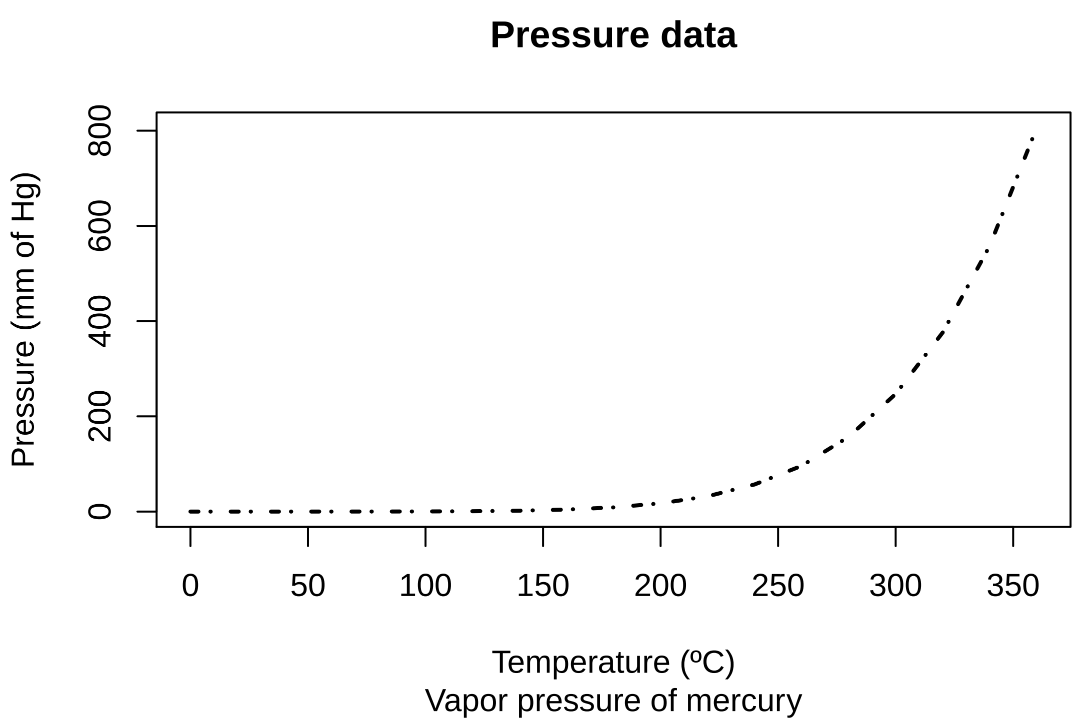


Figure 18: Graph of pressure against temperature with customized line type

3.1.6 pch

The parameter `pch` controls the shape of points in scatterplots and dotplots. There are 26 standard shapes, numbered from 0 to 25, and 1 is the default. The following graphs presents all the standard shapes. For shapes 0 to 20, the option ‘col’ controls the color. For shapes 21 to 25, the filling color can be chosen with the option `bg` while `col` controls the border color.

```
x <- rep(1:5,5)
y <- rep(6:2,rep(5,5))
plot(x,y,type = 'n', axes = FALSE, ylab = '', xlab = '', ylim=c(0,7))
points(x, y, pch = 0:24,  bg='green', col = 'red')
points(3,1,pch = 25, bg = 'green', col = 'red')
text(x, y, labels = 0:24, pos = 3, offset = 0.6)
text(3,1,labels = 25, pos = 3, offset = 0.6)
```

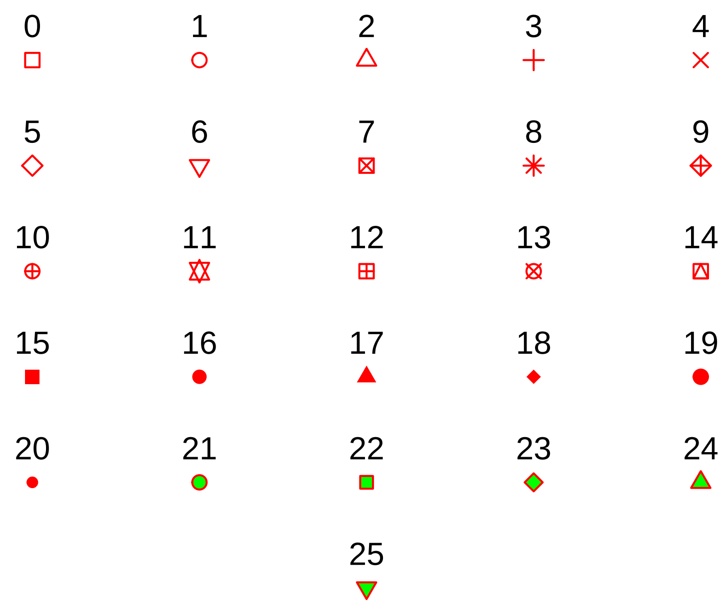


Figure 19: Standard point symbols and their codes

3.2 Color

The option `col` controls the color of the plotting symbol or line. Colors can be specified in different ways. Perhaps the simplest is with the color name within quotation marks (e.g., ‘blue’ or ‘green’). There are hundreds of named colors in R. A list is produced by the function `colors()`. A pdf with the named colors can be found in http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf and a cheatsheet in https://www.nceas.ucsb.edu/sites/default/files/2020-04/colorPaletteCheatsheet.pdf.

Also, eight particular colors can be set by numbers. These colors form the color palette that can be accessed by numbers. The command `palette()` lists the colors in the active palette. Numbers wrap around after 8, so 9 is equal to 1. These colors are

```
## [1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"
## [8] "gray"
```

Figure 20: Colors by numbers in R

The colors in the palette can be modified using the function `palette()` with an argument. One easy way to do this is to use `rainbow` to create a new selection of colors, although this palette has been criticized for giving perceptual problems. Other alternatives can be found in the help for `rainbow`. The following graph shows the palette produce by `rainbow(50)`:

```
pie(rep(1, 50), col = rainbow(50))
```

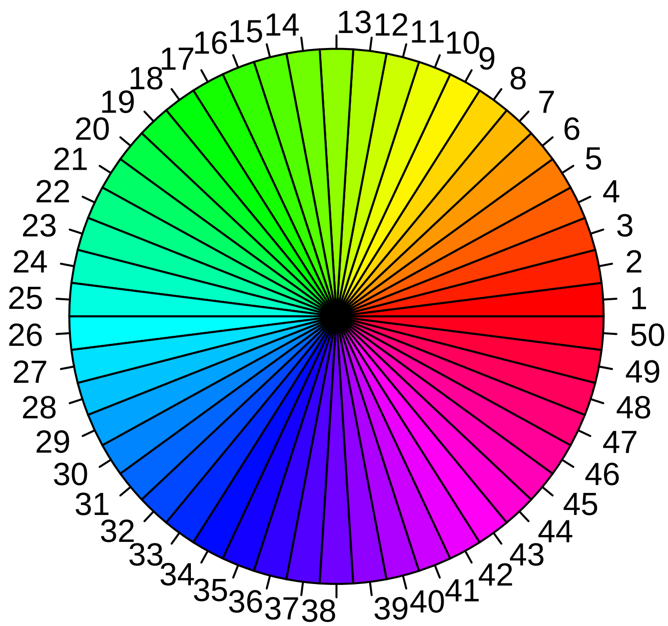


Figure 21: Palette of colors obtained with the function rainbow(50)

Colors can also be specified using their RGB (for Red, Green, and Blue) components with a string of the form '`#RRGGBB`' where each of the pairs '`RR, GG, BB`' consist of two hexadecimal digits giving a value in the range `00` to `FF`.

4 `curve`

This command produce a graph of a single variable function in a given interval. The syntax is

```
curve(expr, from, to, add = FALSE, ...)
```

where

- `expr` is a function with variable `x`
- `from, to` Interval limits
- `add` Logical. If `TRUE` the graph is added to the active window.

The next figure shows a plot of $x^3 - 3x^2$ and $x^2 - 2$ with different colors and line types

```
curve(x^3 - 3*x, -2,2,lwd=2, col='darkblue')
curve(x^2 -2 , add = TRUE, col = 'red3', lwd = 3, lty = 3)
```

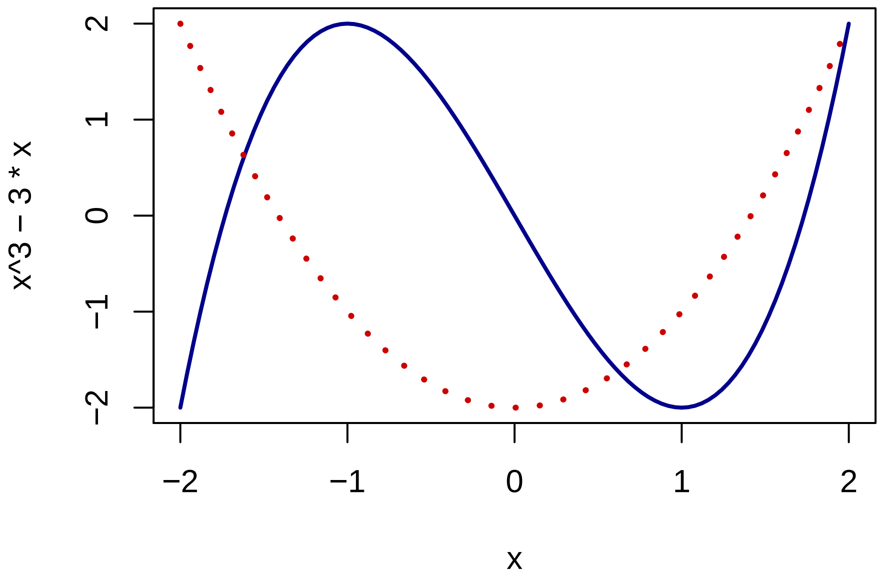


Figure 22: Graphs of $x^3 - 3x^2$ (blue) and $x^2 - 2$ (red dotted line)

5 boxplot

Boxplots were proposed by John W. Tukey in the 1970s as a quick way to visualize the main features of a data set. There are several versions, but in general the interquartile range (iqr) is represented by a box or rectangle, so that the ends of the rectangle are located in the first and third quartiles, as shown in Figure 23. Inside the rectangle, the location of the median is indicated by a line or point. Outside the rectangle, two segments are drawn, called 'whiskers', which reach the furthest data point at a distance less than or equal to $1.5 \times$ (iqr) from the rectangle. Any point that is not included in this range is represented individually and is a potential outlier.

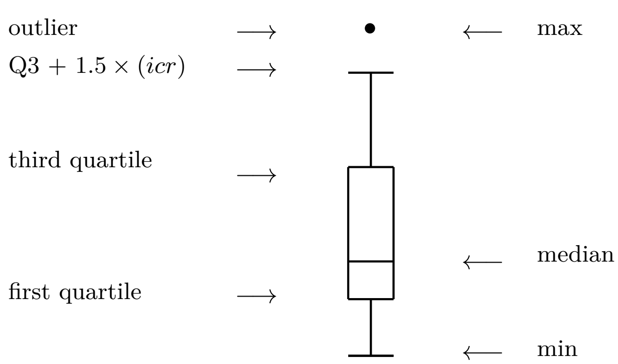


Figure 23: Scheme of a boxplot

The following graph presents boxplots for the four numerical variables in the dataset `iris`.

```
boxplot(iris[,1:4], col='lightblue')
```

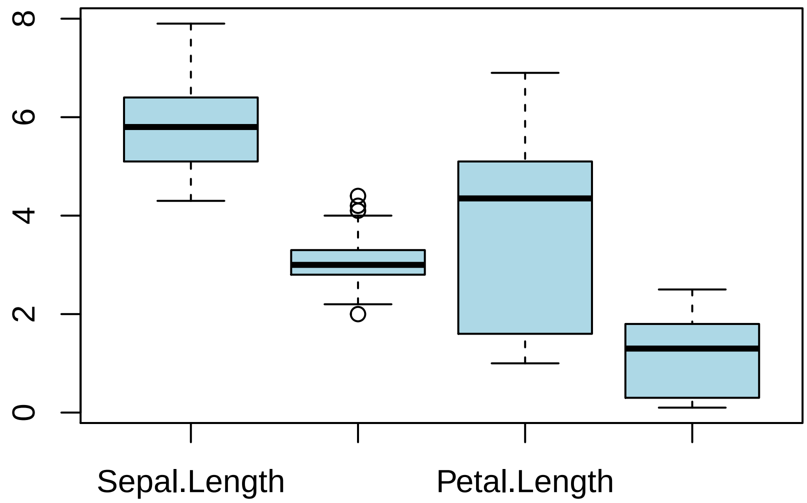


Figure 24: Boxplot for the four numerical variables in the dataset iris

The following figure has boxplots for `Sepal.Length` by `Species` for the `iris` dataset. The option `notch` is set to `TRUE`, which produces the notched boxplots in the figure. Notches, the narrowing of the box near the median, give a rough indication of the significance of the difference between medians. If the notches for two boxplots do not overlap, there is some evidence that the medians differ. Observe that in this example all the notches are disjoint.

```
boxplot(Sepal.Length~Species, data=iris, notch = T, col='lightblue')
```

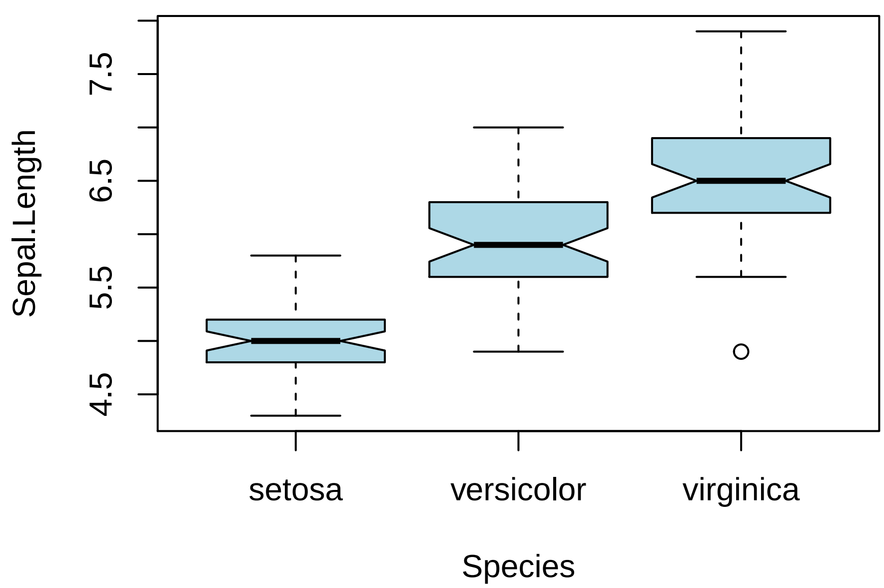


Figure 25: Notched boxplot for sepal length for each of the three species

6 `barplot`

A barplot is a bar diagram for counts. We give examples combining it with `table` to count the number of occurrences of data values. The data counts the repeated values of a simulation of 100 Poisson random variables with parameter 5. The first plot has all bars in grey while the second uses colors from a `rainbow` palette.

```r
tN <- table(Ni <- rpois(100, lambda=5))
barplot(tN)
```

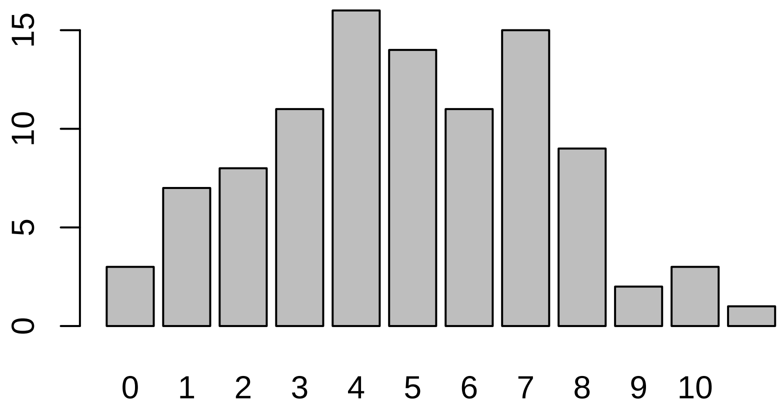


Figure 26: Barplot for the values of 100 simulated Poisson random variables with parameter 5

```r
barplot(tN, col=rainbow(20))
```

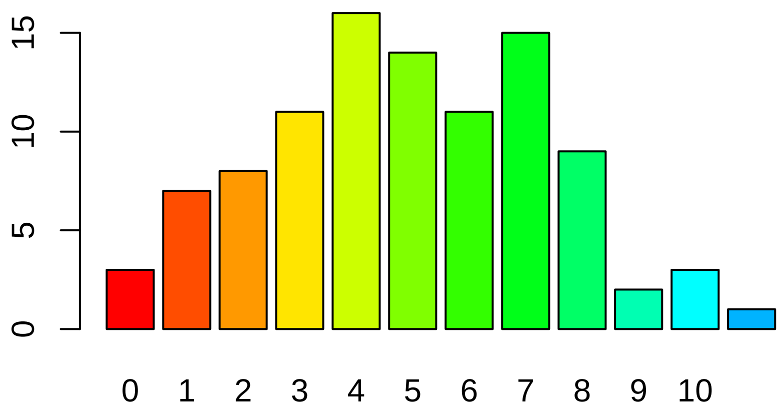


Figure 27: Barplot for the values of 100 simulated Poisson random variables with parameter 5. Colors for the bars come from a rainbow palette

7 Histograms

We will use the `morley` dataset that contains the results of (one of) Michelson's experiments on the speed of light to show examples of the use of the function `hist`. There are five experiments with 20 consecutive runs each in the dataset. The value corresponds to the observed experimental value after subtracting 299000 km/s.

We start by attaching the `morley` dataset.

```r
str(morley)
```

```
## 'data.frame':    100 obs. of  3 variables:
##  $ Expt : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Run  : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Speed: int  850 740 900 1070 930 850 950 980 980 880 ...
```

```r
attach(morley)
```

We first graph a scatterplot of the consecutive runs. Different colors are used for the five experiments.

```
plot(Speed,col=Expt, pch=19)
```

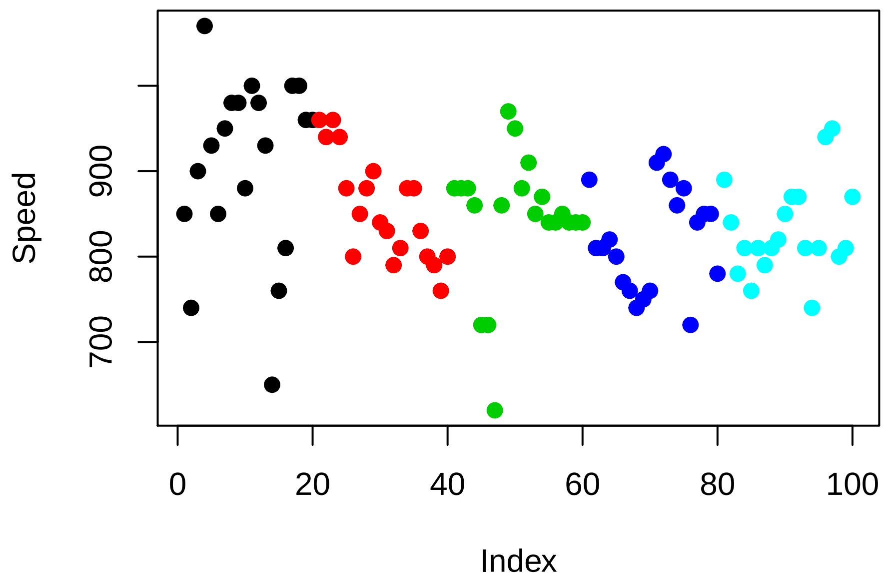


Figure 28: Scatterplot for the 100 runs in Michelson's speed of light experiment.

A plain histogram of the data using the `hist` function:

```
hist(Speed)
```

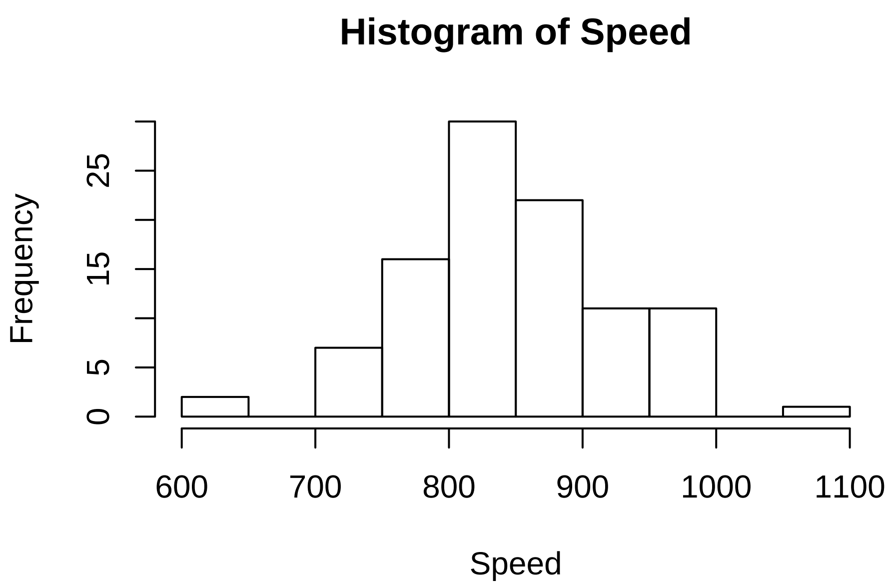


Figure 29: Histogram for the 100 measurements of the speed of light in Michelson's experiment

For the next version of the histogram, we use some options to modify the resulting plot, we specify 20 breaks (this works only as a suggestion that in many cases is not followed by the plotting function). The `probability = T` option produces a graph of relative frequencies, but the only change is in the units in the y axis. We also add a title, color and custom axis labels

```r
hist(Speed, breaks = 20, probability = T, col = 'azure2', xlab='Speed',
     ylab='Relative Frequency', main = "Michelson's Experiment")
```

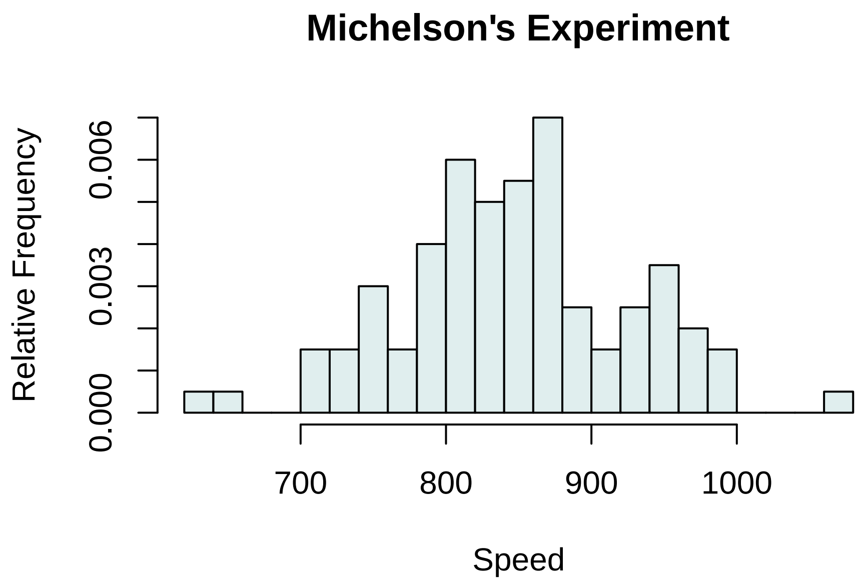


Figure 30: Histogram for the 100 measurements of the speed of light in Michelson's experiment

The following figure has a histogram for each experiment plus a histogram for all the experiments together.

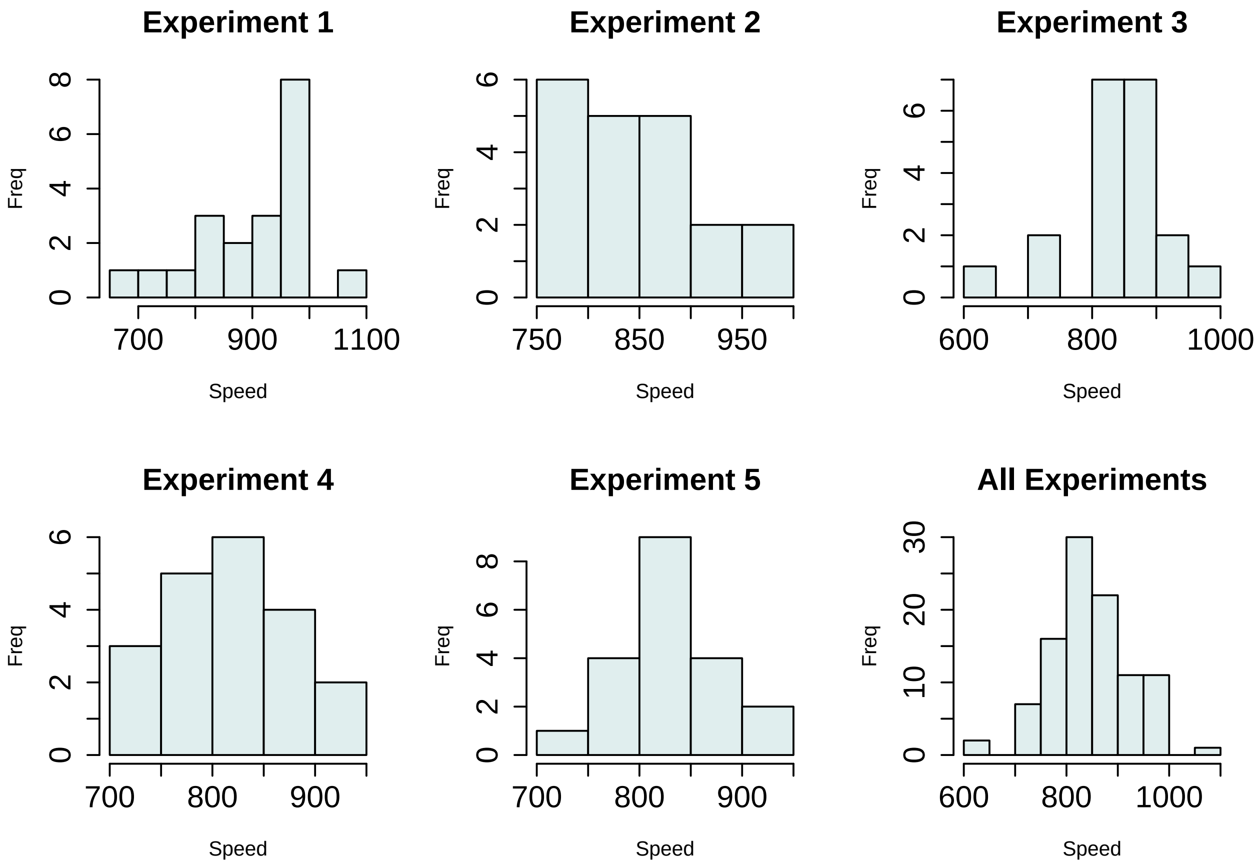


Figure 31: Histograms for the five experiments

The histograms in the previous figure have different scales in the x-axis. The reason for this is that it is the range of values to be plotted in each graph what determines the scales, and since the values differ in this case, the scales are also different. However, this may be misleading if we want to use the graphs to compare the results. For this reason we plot again the graphs with a common scale in the x-axis, which is set by hand using the option `xlim`.

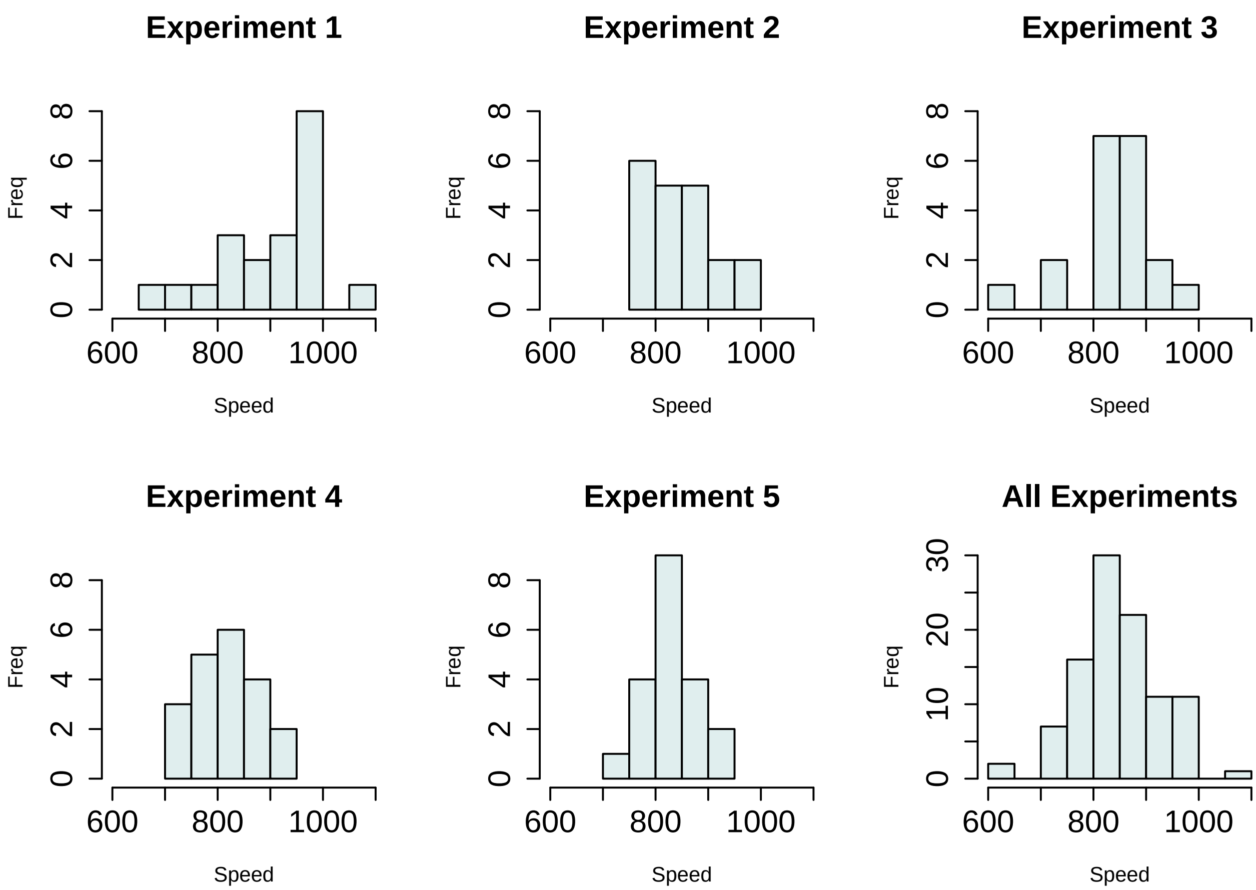


Figure 32: Histograms for the five experiments

7.1 Number of breaks

The number of bins in a histogram may have an important visual effect, that may affect the conclusions the viewer draws from the graph. One of the tools for fixing this number is the option `breaks` that can be, among others, a vector with the endpoints for the bins, a function to compute these endpoints or a single number giving the number of bins for the histogram. In the latter case the number is only a suggestion and will not be necessarily followed. As an example, the following three graphs use the default value plus two different options and the results is always the same.

```
par(mfrow = c(1,3))
hist(Speed,col = 'azure2')
hist(Speed, breaks = 8,col = 'azure2')
hist(Speed, breaks = 16,col = 'azure2')
```

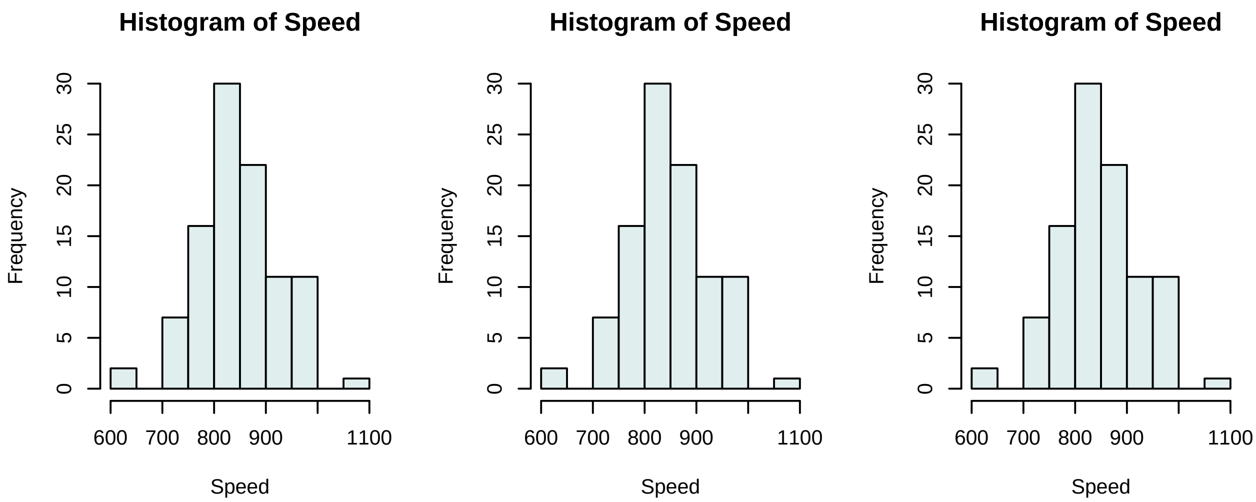


Figure 33: Histogram for the 100 measurements of the speed of light in Michelson's experiment with three different options for the number of bins

7.2 Anchor

By using again the `breaks` option but this time with a vector that gives the exact breakpoints, we can explore the effect of the anchor, which is the starting point for the set of bins used to define the histogram. Even with the same number of bins, simply by varying the exact location of the intervals we may see changes in the histograms that give the viewer different perceptions abouth the characteristics of the data set.

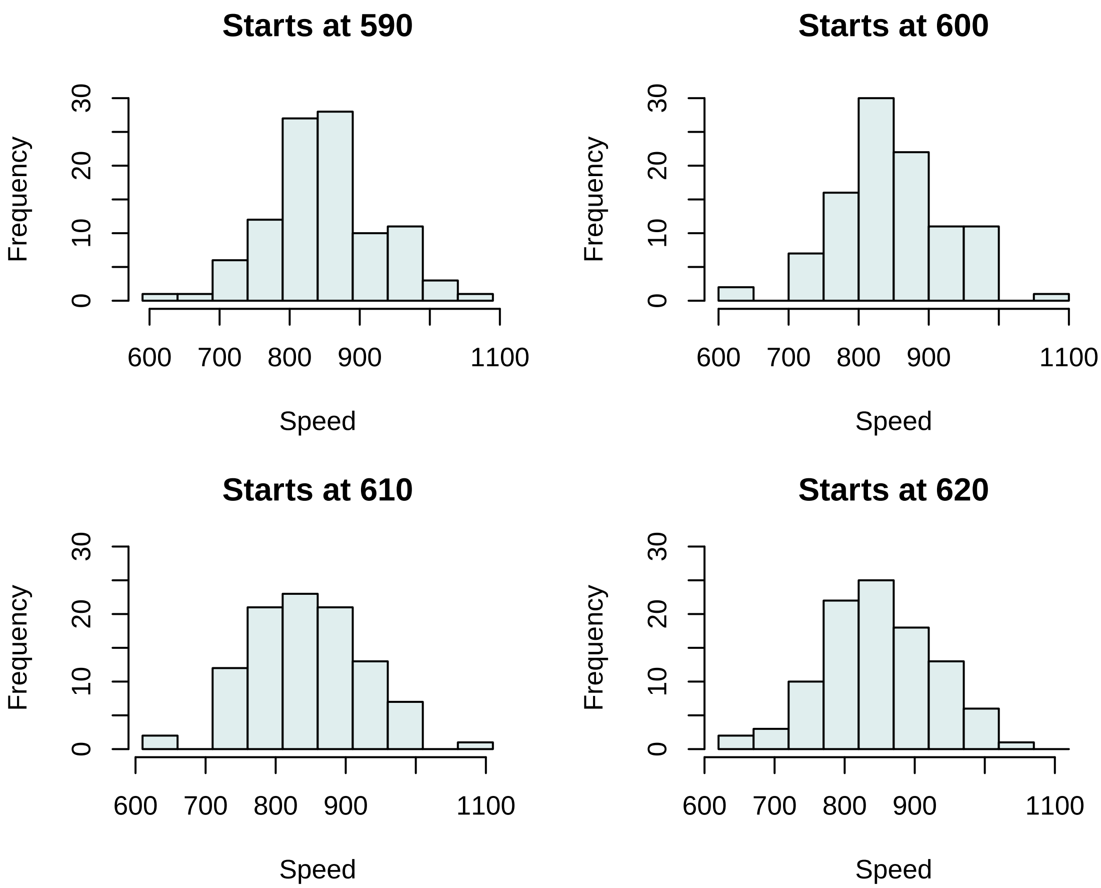


Figure 34: Histogram for the 100 measurements of the speed of light in Michelson's experiment with four different starting points or anchors

8 `dotchart` and `pie`

Dotcharts or dotplots were proposed by W.S. Cleveland as an alternative to pie charts. It has been shown experimentally that the human eye is not very good at judging relative angles and areas and is good at comparing distances on a common scale. Dotcharts can always be used instead or pie charts for showing data.

The following graph shows two sets of data as dotcharts and pie charts, for comparison purposes. The data plotted are the mean and standard deviation for the five speed of light experiments.

```
mu <- numeric(5); std.dev <- numeric(5)
for (i in 1:5) { mu[i] <- mean(Speed[Expt==i])
  std.dev[i] <- sd(Speed[Expt==i])}
par(mfrow=c(2,2))
dotchart(mu, labels = 1:5, pch=19, main = 'Average')
dotchart(std.dev, labels = 1:5, pch=19, main = 'Standard dev.')
par(mar=c(4.5,3,3,1))
pie(mu, labels = 1:5, main = 'Average', radius = 1)
pie(std.dev, labels = 1:5, main = 'Standard dev.', radius = 1)
```

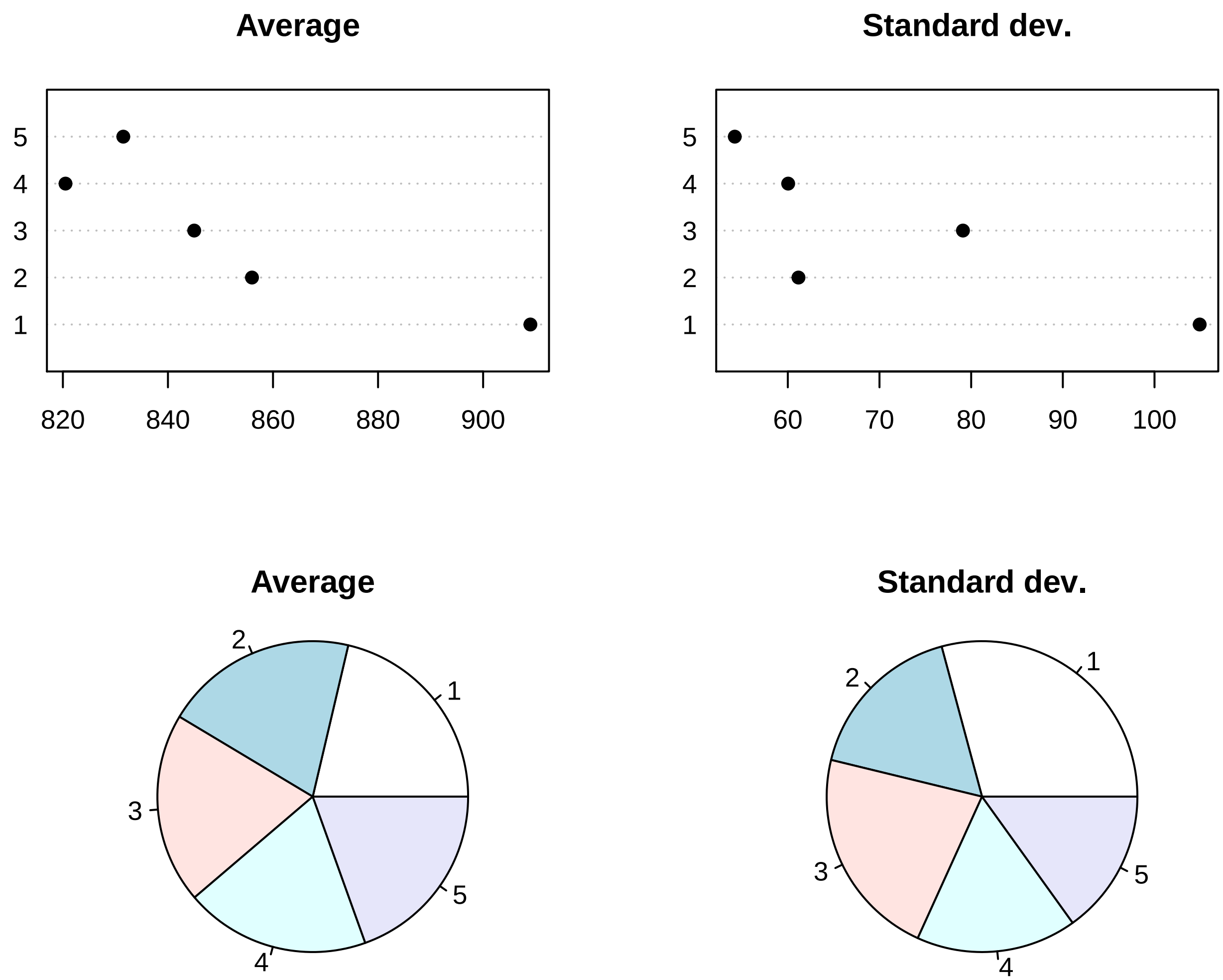


Figure 35: Dotchart and pie chart for the average and standard deviation for the speed of light experiments of Michelson

The following pie charts present three examples where the values to be plotted are close to each other. Try to determine, in each case, the correct order.

```
cols <- hcl.colors(15, "Set 2")
dat1 <- c(19,19.5,20,20.5,21)
dat2 <- rev(dat1)
dat3 <- c(19,20,21,20.5,19.5)
par(mfrow = c(1,3))
pie(dat1,labels = 1:5,radius = 1, col = cols[c(1,6,8,10,14)])
pie(dat2,labels = 1:5,radius = 1, col = cols[c(1,6,8,10,14)])
pie(dat3,labels = 1:5,radius = 1, col = cols[c(1,6,8,10,14)])
```

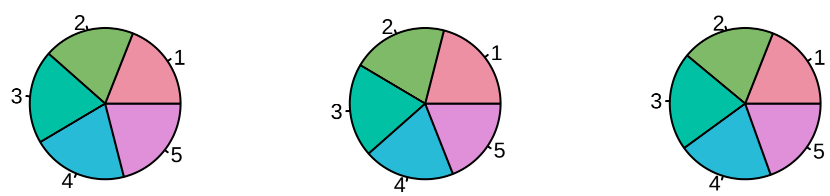


Figure 36: Pie charts for three data sets

It is very difficult to determine relative sizes in a pie chart. On the other hand, if we draw a barplot,

```
par(mfrow = c(1,3))
barplot(dat1, col = cols[c(1,6,8,10,14)], yaxt = 'n',names.arg = 1:5)
barplot(dat2, col = cols[c(1,6,8,10,14)], yaxt = 'n',names.arg = 1:5)
barplot(dat3, col = cols[c(1,6,8,10,14)], yaxt = 'n',names.arg = 1:5)
```

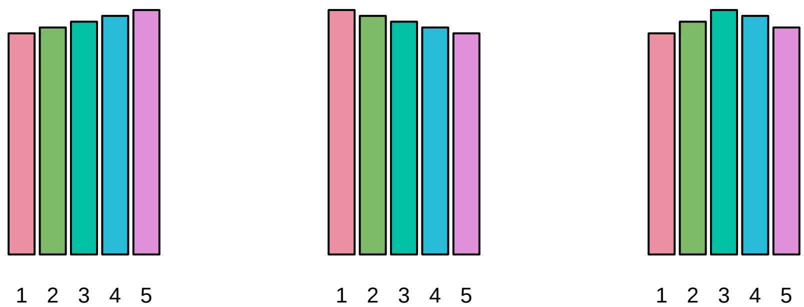


Figure 37: Bar charts for three data sets

```
par(mfrow=c(1,1))
```

the order is very clear in all cases. Dotcharts are also a good choice in this situation.

```
oldpar <- par(no.readonly = TRUE) #
par(xaxt = 'n', mfrow = c(1,3))
dotchart(dat1,1:5, col = cols[c(1,6,8,10,14)], pch = 16,cex = 1.2, xlim = c(18.5,21.5))
dotchart(dat2,1:5, col = cols[c(1,6,8,10,14)], pch = 16,cex = 1.2, xlim = c(18.5,21.5))
dotchart(dat3,1:5, col = cols[c(1,6,8,10,14)], pch = 16,cex = 1.2, xlim = c(18.5,21.5))
```

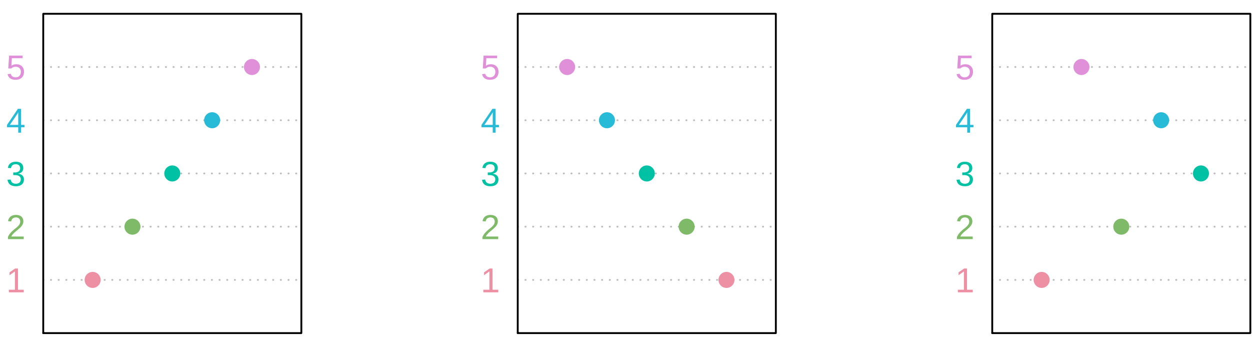


Figure 38: Dotcharts for three data sets

```
par(oldpar)
```

9 pairs

This produces a matrix of graphs. We give three examples using the dataset `iris`.

```
pairs(iris)
```

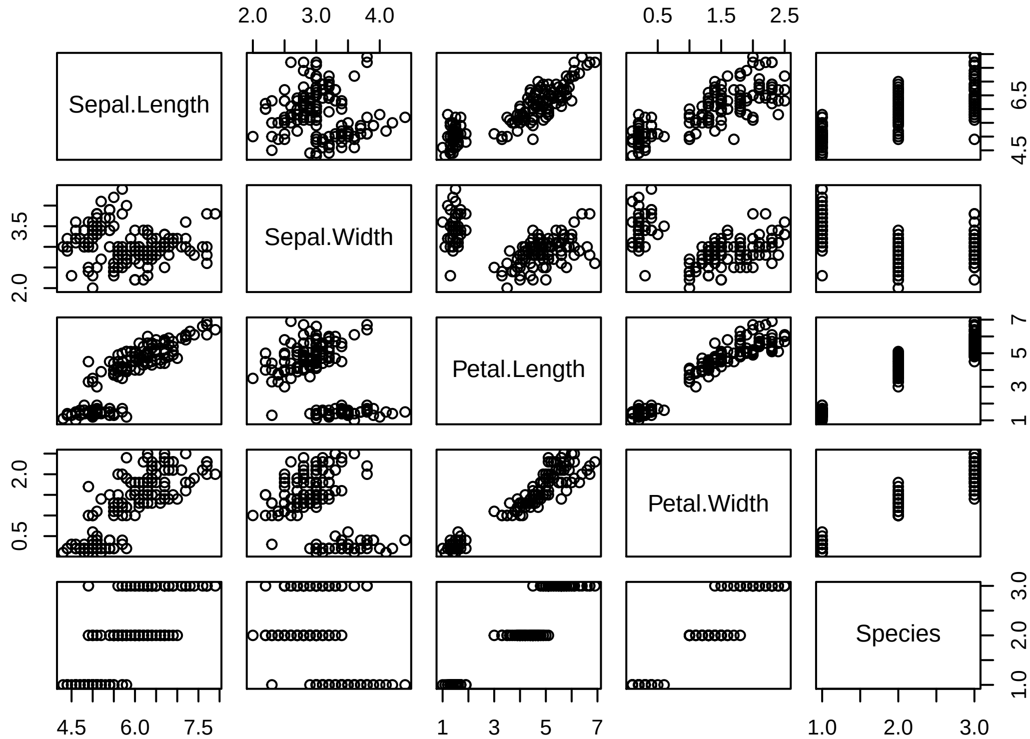


Figure 39: Matrix of graphs for the iris dataset using the function pairs

```
pairs( ~ Sepal.Length + Petal.Length + Petal.Width)
```

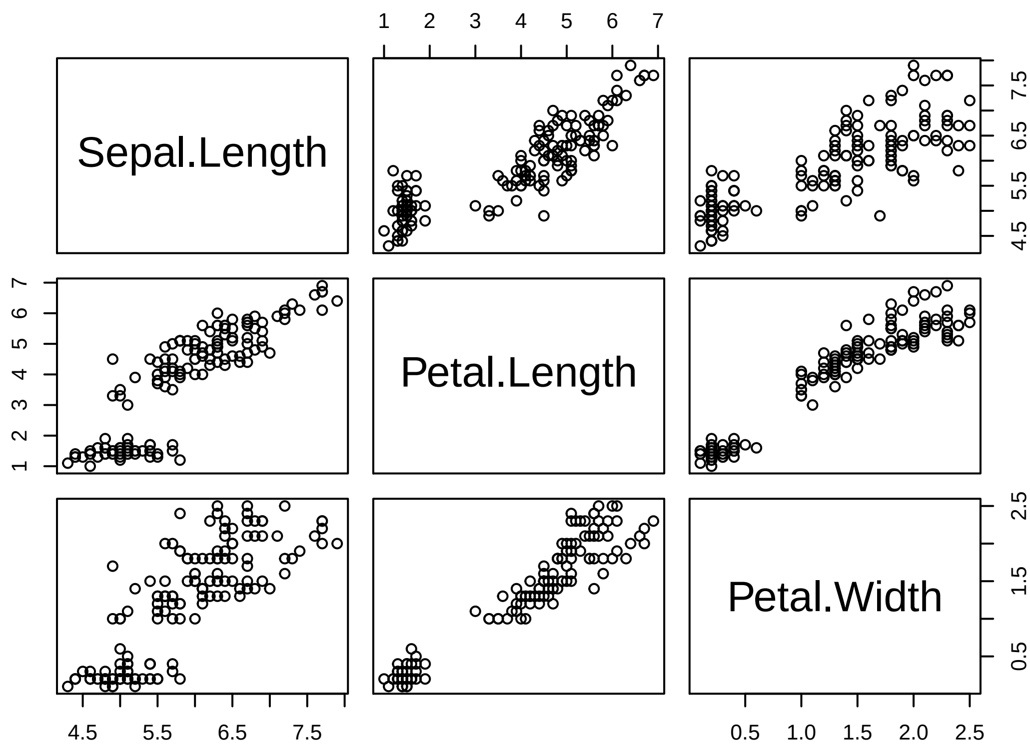


Figure 40: Matrix of graphs for three of the variables in the iris dataset using the function pairs

```
pairs(iris[1:4], main = "Anderson's Iris Data",pch = 21,
  col = c('red','green3','blue')[iris$Species])
```

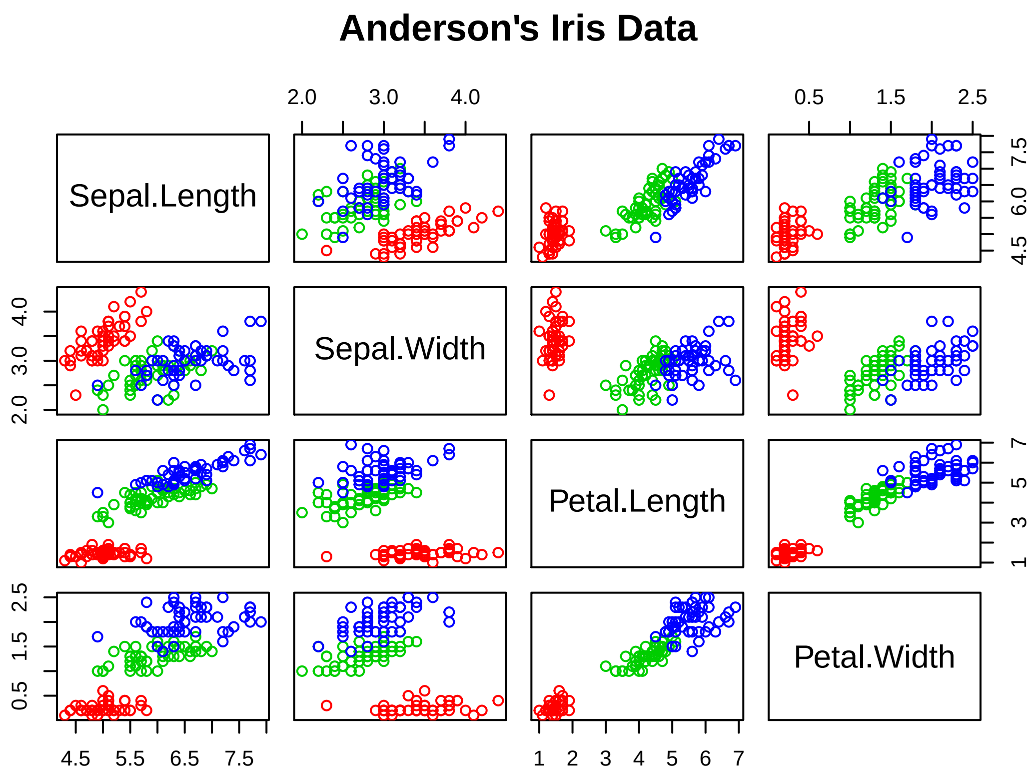


Figure 41: Matrix of graphs for the numerical variables in the iris dataset using the function pairs, colores by species

10 Three-dimensional graphs

The `base` package in `R` has several function for handling three dimensional plots. `contour` and `filled.contour` give two-dimensional representations of a three-dimensional surface using contour lines. We give examples using two-dimensional Gaussian densities.

10.1 `contour` and `filled.contour`

Creates a contour plot or adds contour lines to an existing plot. Syntax:

```
contour(x = seq(0, 1, length.out = nrow(z)),
        y = seq(0, 1, length.out = ncol(z)),
        z, nlevels = 10, add = FALSE)
```

10.1.1 Bivariate normal distribution.

```
library(mvtnorm)
x.points <- y.points <- seq(-3,3,length.out=100)
z <- matrix(0,nrow=100,ncol=100)
mu <- c(1,1); sigma <- matrix(c(2,1,1,1),nrow=2)
for (i in 1:100) {
  for (j in 1:100) {
```

```
  z[i,j] <- dmvnorm(c(x.points[i],y.points[j]), mean=mu,sigma=sigma)
  }
}
contour(x.points,y.points,z)
```

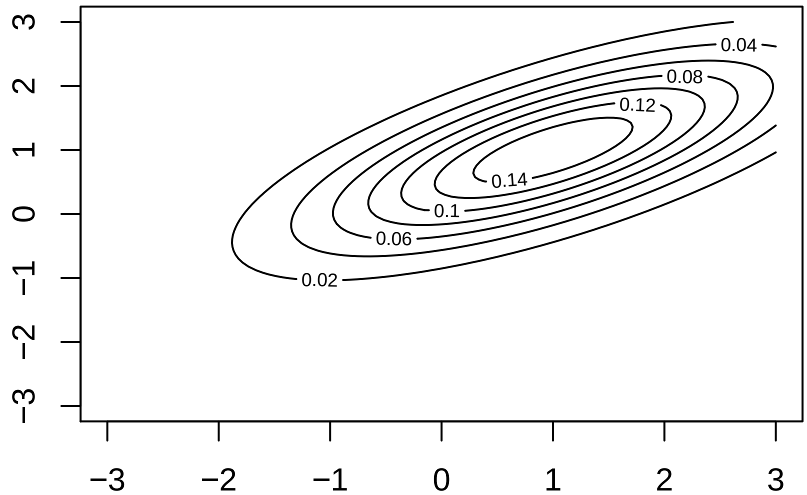


Figure 42: Colored contour plot for a bivariate Gaussian density

```
filled.contour(x.points,y.points,z)
```

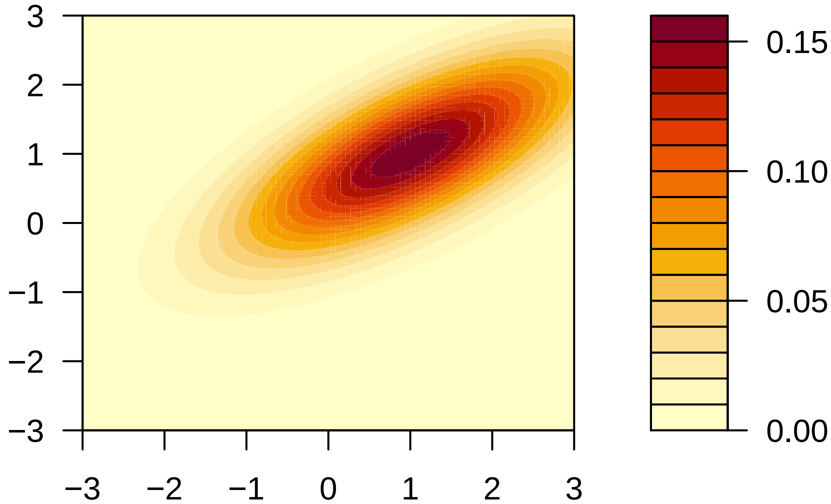


Figure 43: Colored contour plot for a bivariate Gaussian density

```
library(mvtnorm)
normal.contour <- function(mu=c(0,0),
          sigma=matrix(rep(1,4),nrow=2)){
  x.points <- y.points <- seq(-3,3,length.out=100)
  z <- matrix(0,nrow=100,ncol=100)
  for (i in 1:100) {
    for (j in 1:100) {
      z[i,j] <- dmvnorm(c(x.points[i],y.points[j]),
```

```
                        mean=mu,sigma=sigma)
    }
  }
  contour(x.points,y.points,z, main=paste('Corr = ', sigma[1,2]),
          drawlabels = FALSE)
  abline(h=0); abline(v=0)
}
mu = c(0,0)
sigma1 <- matrix(c(1,0,0,1),nrow=2); sigma2 <- matrix(c(1,.25,.25,1),nrow=2)
sigma3 <- matrix(c(1,.5,.5,1),nrow=2); sigma4 <- matrix(c(1,.75,.75,1),nrow=2)
par(mfrow=c(2,2))
normal.contour(mu,sigma1); normal.contour(mu,sigma2)
normal.contour(mu,sigma3); normal.contour(mu,sigma4)
```

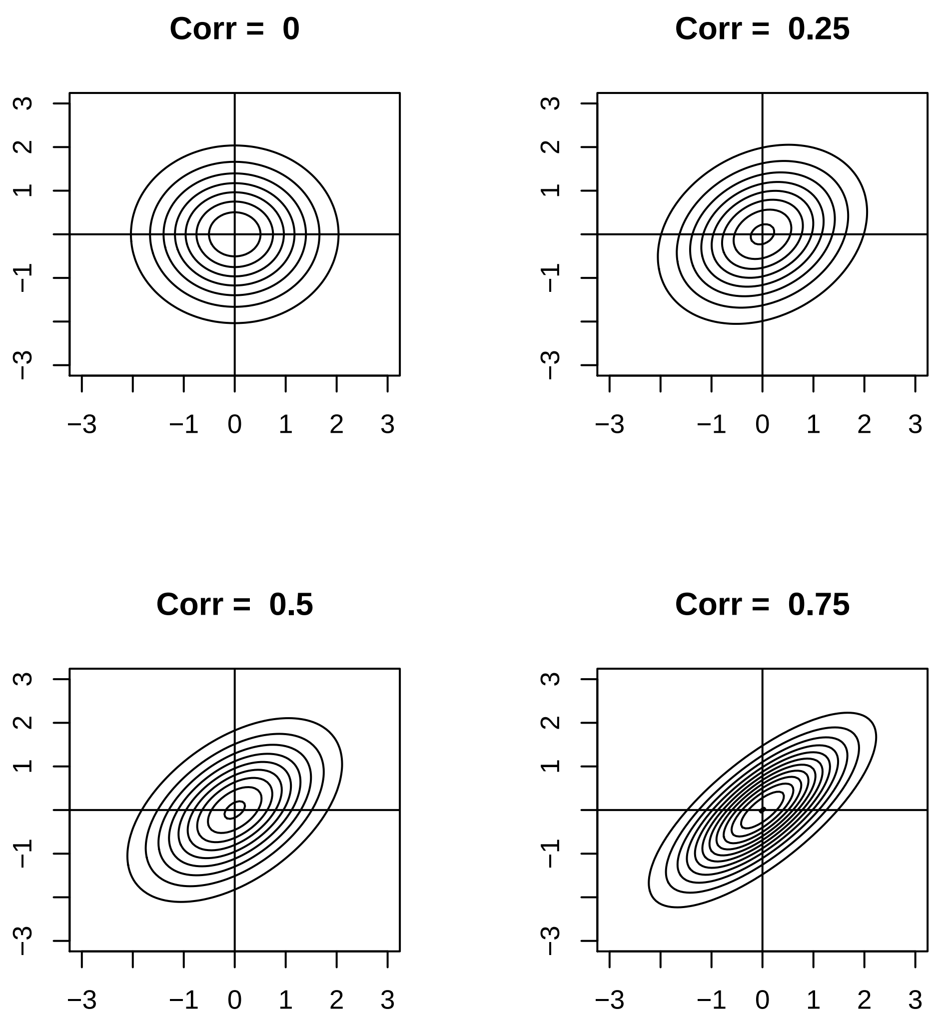


Figure 44: Contour plots for four bivariate Gaussian density with different correlations

10.2 `image`

This function can be used to display three dimensional data as a two-dimensional grid of rectangles with colors corresponding to the values of z.

```r
image(x.points,y.points,z, xlab = '', ylab = '')
```

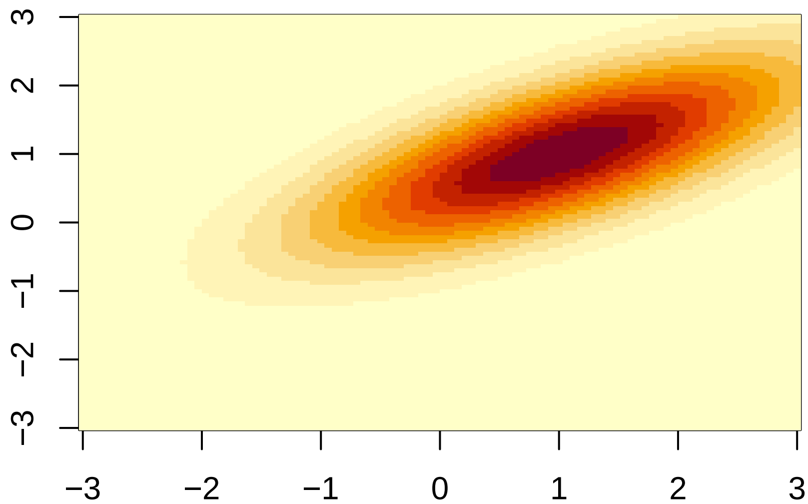
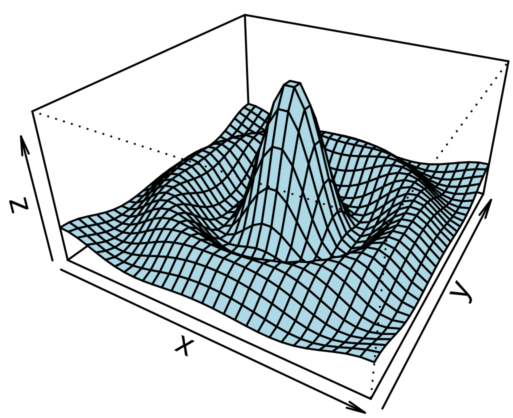


Figure 45: Image of a bivariate Gaussian density

10.3 persp

This function displays a perspective plot of a surface over the xy plane.

```r
y <- x <- seq(-10, 10, length= 30)
f <- function(x, y) { r <- sqrt(x^2+y^2); 10 * sin(r)/r }
z <- outer(x, y, f); z[is.na(z)] <- 1
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
```



11 Other Chart Types

The high level plotting functions that follow are not commonly used. We only give a very brief description with examples taken form the help files.

11.1 sunflowerplot(x,y)

This function plots multiple points as 'sunflowers' with as many leaves as common values there are.

```r
sunflowerplot(iris[, 3:4], size = .07)
```

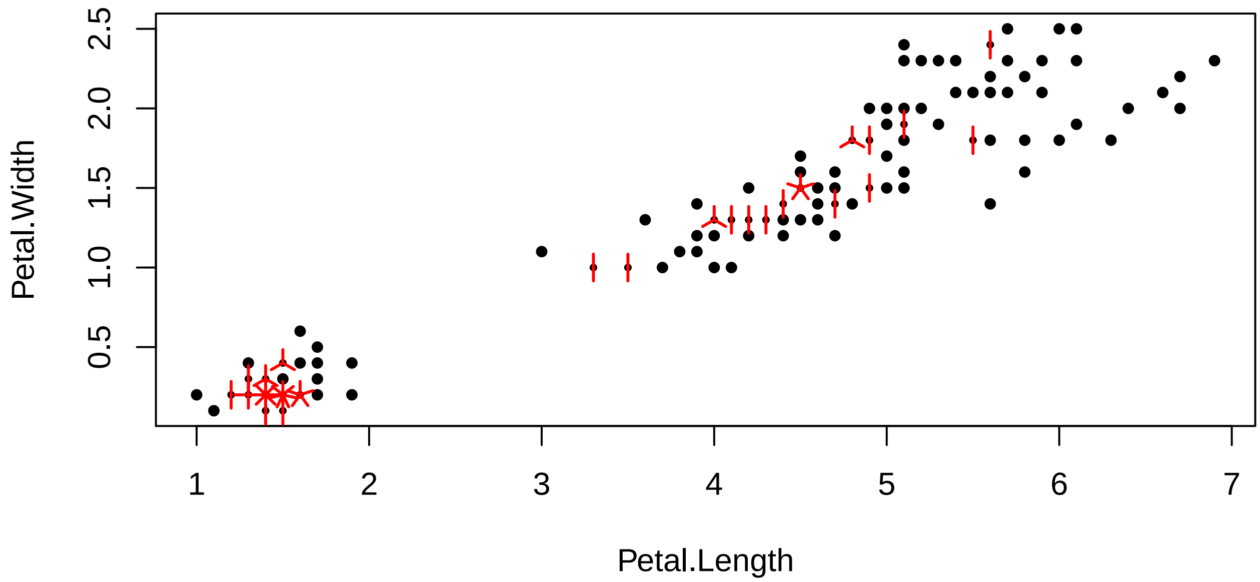


Figure 46: Example of a sunflowerplot

11.2 `stripchart(x)`

Produces a one-dimensional dot plot of the data.

```
x <- rnorm(50); xr <- round(x, 1)
stripchart(x) ; m <- mean(par("usr")[1:2])
text(m, 1.04, "stripchart(x, \"overplot\")")
stripchart(xr, method = "stack", add = TRUE, at = 1.2)
text(m, 1.35, "stripchart(round(x,1), \"stack\")")
stripchart(xr, method = "jitter", add = TRUE, at = 0.7)
text(m, 0.85, "stripchart(round(x,1), \"jitter\")")
```

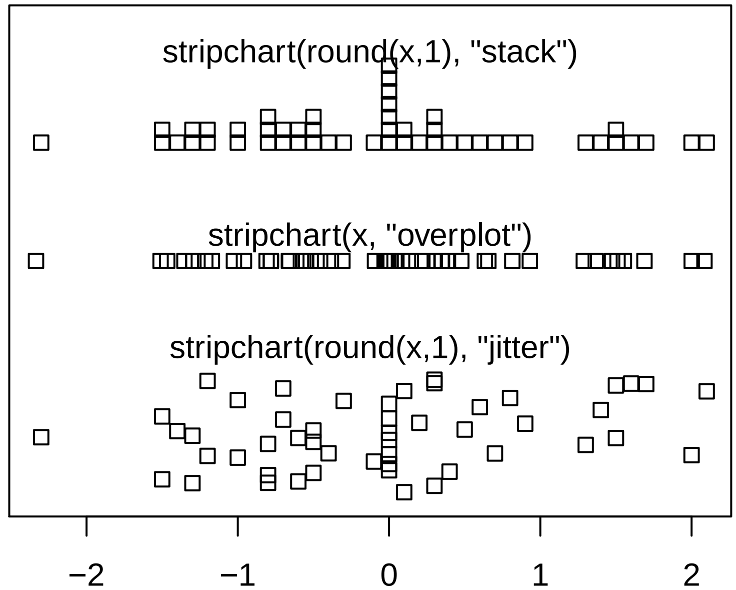


Figure 47: Example of a ‘stripchart‘

11.3 `stars(x)`

This function plots multivariate data using stars, where each 'ray' of the star corresponds to one dimension.

```
stars(iris[,1:4], key.loc=c(25,20), nrow=6, len=2,
      main= 'Star plots for individual plants, iris dataset',
      col.stars=as.numeric(Species))
```

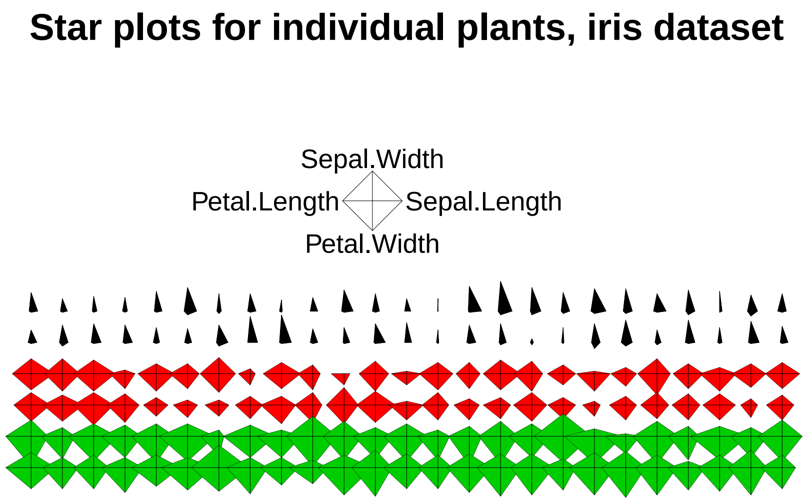


Figure 48: Example of a 'star' plot

12 Legends

The command `legend` adds a legend to a plot in `R`. Syntax:

```
legend(x, y = NULL, legend, fill, col, bg)
```

- `x` and `y`: co-ordinates to position the legend
- `legend`: text of the legend
- `fill`: colors to use for filling the boxes with the legend text
- `col`: colors of lines and points in the legend text
- `bg`: the background color for the legend box.

The legend position may be specified setting `x` to one of the following: "`bottomright`", "`bottom`", "`bottomleft`", "`left`", "`topleft`", "`top`", "`topright`", "`right`" and "`center`".

```
with(iris, plot(Petal.Length,Petal.Width, pch=16+as.numeric(Species),
     col=c('red2','green3','blue3')[iris$Species]))
legend('topleft',legend=c('setosa','versicolor','virginica'),
       col=c('red2','green3','blue3'), pch=16+(1:3))
```

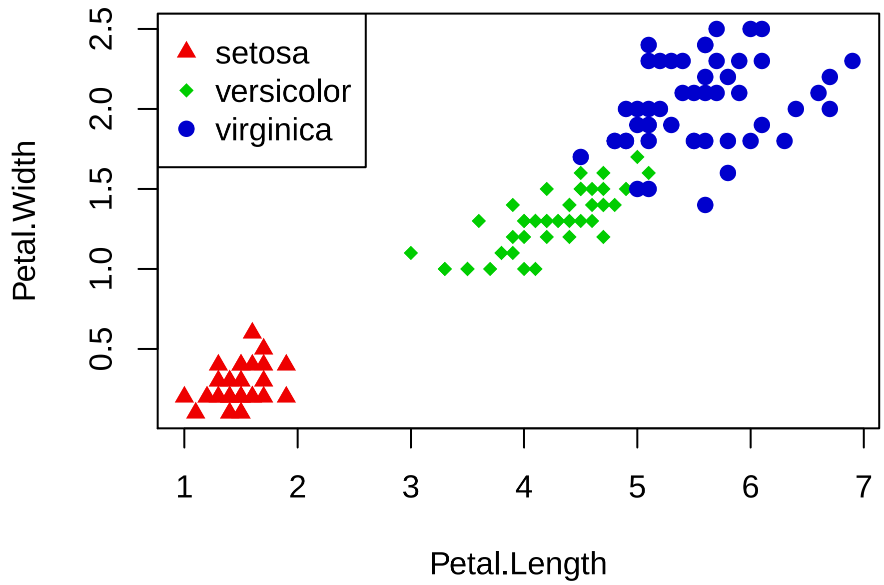


Figure 49: Example of the use of legend

```
boxplot(mpg~cyl, xlab="Cylinders", ylab="Miles/gallon", col=topo.colors(3))
legend("bottomleft", inset=.02, title="Number of Cylinders",
   c("4","6","8"), fill=topo.colors(3), horiz=TRUE, cex=0.8)
```

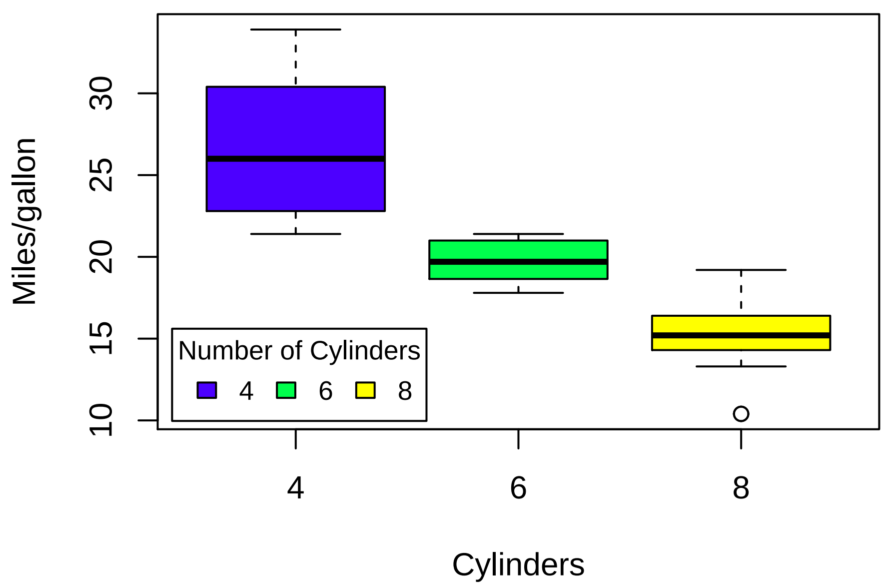


Figure 50: Example of the use of legend

13 Low Level Commands

So far we have looked at high level plotting commands in R, which are the functions that produce graphs. Low level commands are useful to add information to existing plots.

13.1 points

The function `points` adds points to an active graph. As an example we add points to a boxplot. Since there are many repeated values that overlap in the graph, we use the function `jitter` to add some noise on the horizontal direction, to be able to tell these points appart.

```
boxplot(Petal.Length ~ Species, data = iris)
points(jitter(as.numeric(Species), factor = 0.2), Petal.Length,
       col = cols[c(6,8,10)][as.numeric(Species)])
```

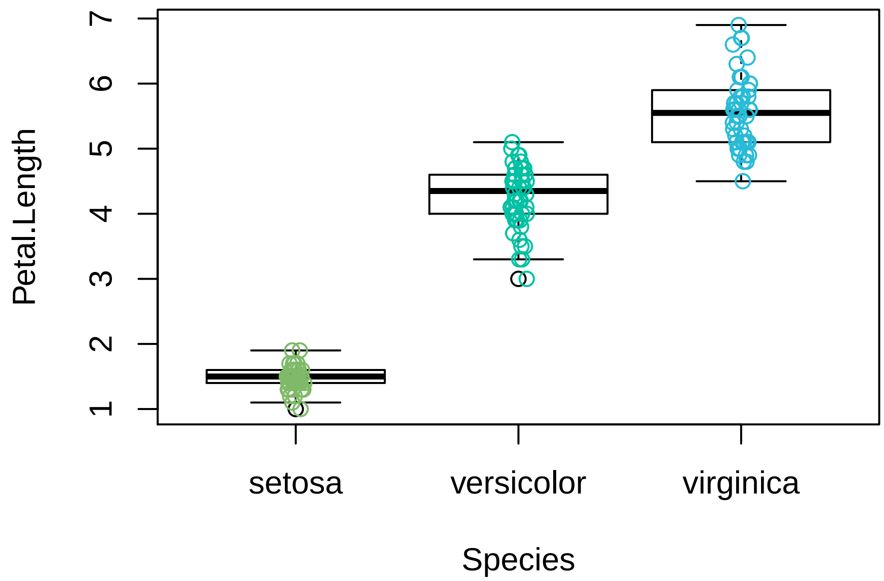


Figure 51: Boxplots of petal length as a function of species for the iris dataset. Points were added with the function ‘points()‘

13.2 lines

This function adds lines to an active graph. The dataset `Indometh` has data on concentration (mcg/ml) of the compound indometacin as a function of time for six subjects. We want to make a graph of the six concentration curves using different colors for each subject. One way to do this is to plot first the curve for subject 1 using `plot` and then add the other curves using `lines` in a `for` loop.

```
plot(conc[Subject == 1] ~ time[Subject == 1], data = Indometh,col=cols[2],
     type = 'o', ylim = c(0,2.7), xlab = 'time (hr)',ylab = 'concentration (mcg/ml)',
     main='Concentration of indometacin')
for(i in 2:6){
  lines(conc[Subject == i] ~ time[Subject == i], data = Indometh,
        type = 'o', col = cols[2*i])}
legend('topright', legend = c('1','2','3','4','5','6'), col = cols[2*(1:6)],
        pch = rep(1,6), lwd = rep(1,6))
```

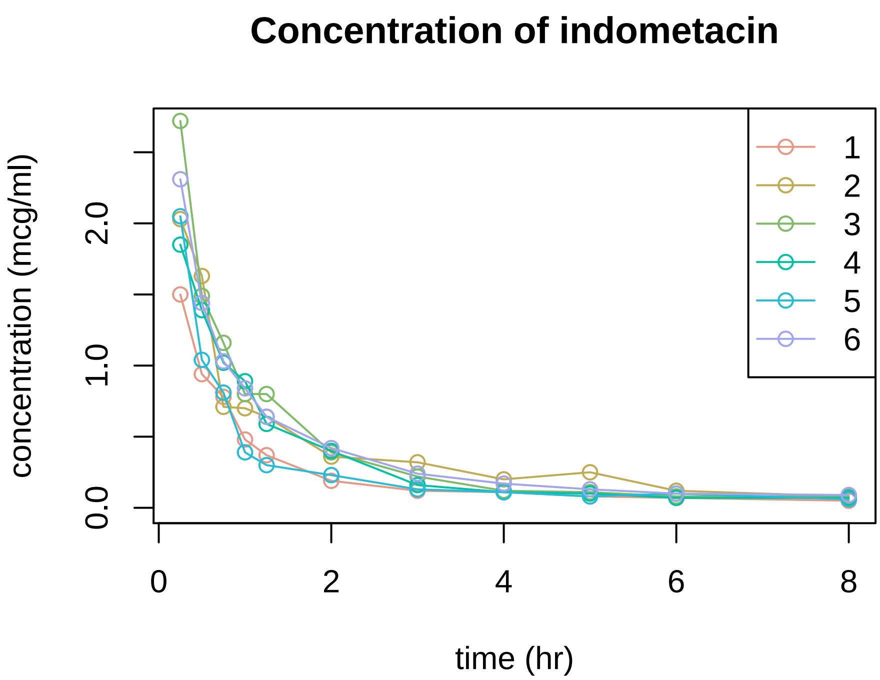


Figure 52: Example of the use of 'lines()'

A related function is `abline`, which draws straight lines. There are several possible syntaxes depending on the way the straight line is defined. Two simple cases are `abline(h=w)` and `abline(v=w)` that draw horizontal and vertical lines, respectively, at the value specified by `w`.

```
plot(cars)
abline(h=60)
abline(v=15)
```

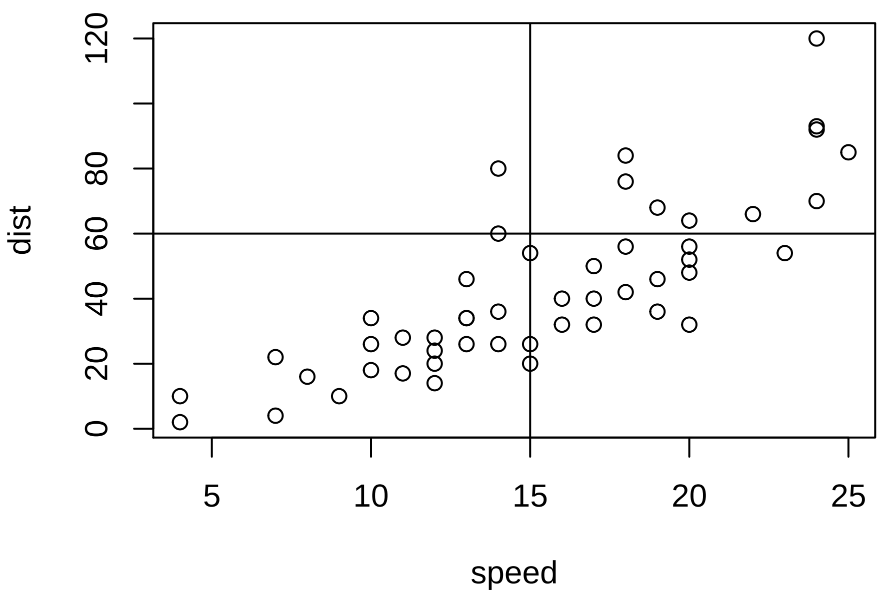


Figure 53: Example of the use of 'abline'

To overplot the regression line on a graph we can combine `abline` with the regression function `lm`,

```
plot(cars)
abline(reg = lm(dist ~ speed, data = cars))
```

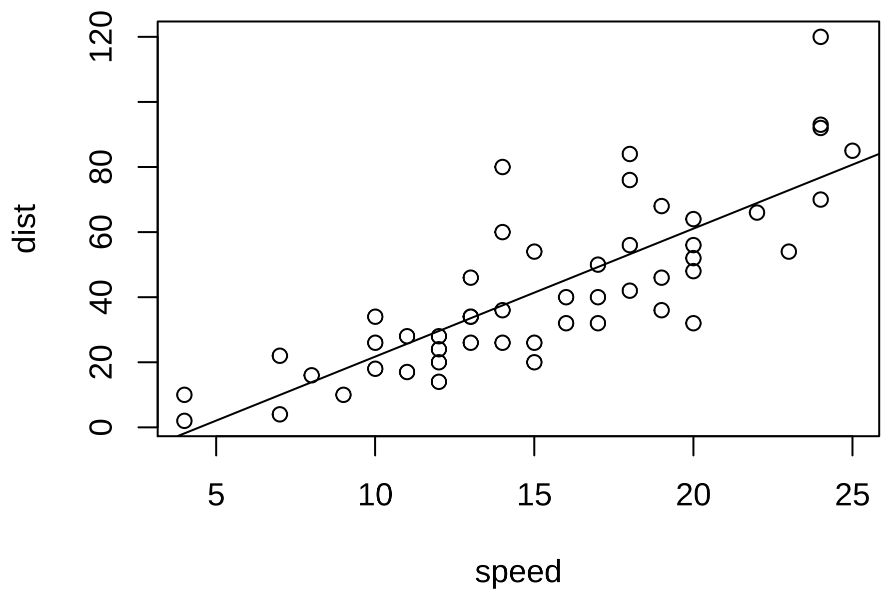


Figure 54: Example of the use of 'abline' to add a regression line to a scatterplot

The function `lines` can also be used with other functions to overlay a smooth curve on top of a set of data points. In the next example we use it in combination with `loess`, which fits a local polynomial to a set of points with up to four dimension (or predictors). We give an example using again the `cars` dataset. The parameter `span` controls the degree of smoothing.

```
cars.lo03 <- loess(dist ~ speed, data = cars, span = 0.3)
cars.lo06 <- loess(dist ~ speed, data = cars, span = 0.6)
cars.lo09 <- loess(dist ~ speed, data = cars, span = 0.9)
plot(cars)
lines(predict(cars.lo03), x = cars$speed, col = 2, lwd = 1.5)
lines(predict(cars.lo06), x = cars$speed, col = 3, lwd = 1.5)
lines(predict(cars.lo09), x = cars$speed, col = 4, lwd = 1.5)
abline(reg = lm(dist ~ speed, data = cars), col = 5)
leg.text <- c('span = 0.3','span = 0.6','span = 0.9','linear reg.')
legend('topleft',leg.text,col=2:5, lwd = 1.5)
```

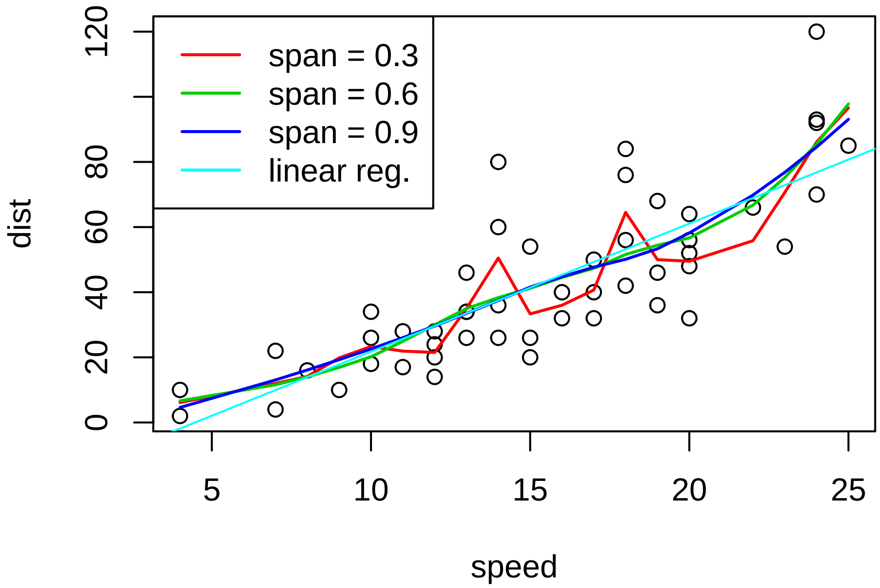


Figure 55: Example of the use of ‘lines‘ and ‘loess‘ to add a smoothing polynomial to a scatterplot

13.3 Other commands

There are many other low level graphical commands that allow the user to modify existing graphs. Some of them are.

- `axis` adds an axis to an existing plot. It is used with the option `axes = FALSE` in the original plot. There are option to specify the side, position, labels and other characteristics.
- `arrows` draws arrows between pairs of points.
- `segments` draws line segments between pairs of points.
- `text` draws strings of characters.
- `polygon` draws polygons with given vertices.

In the next example we use some of these functions. Observe that the initial `plot` function does not draw anything, but it defines the plotting region that will be used by the additional functions. We start by plotting a standard graph of miles per gallon (`mpg`) against displacements (`disp`) for the `mtcars` dataset.

```
with(mtcars,plot(disp, mpg))
```

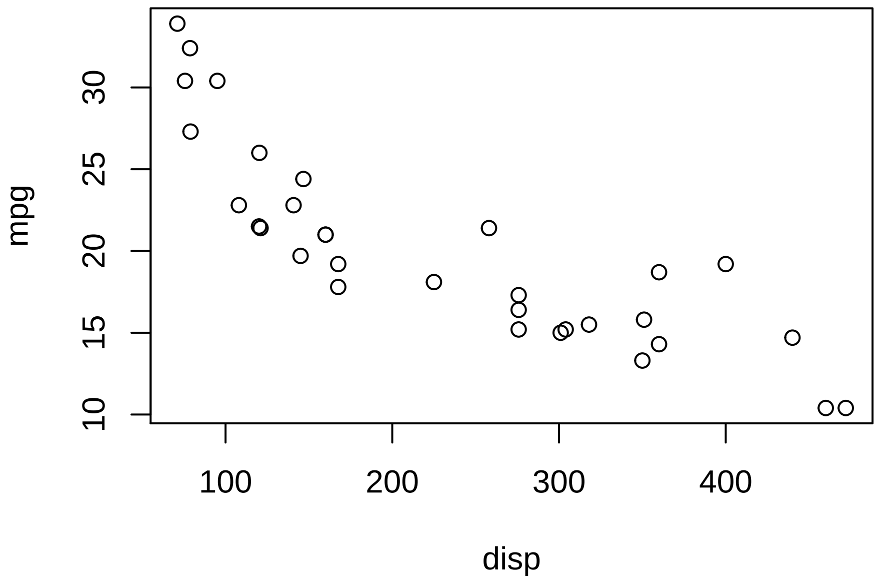


Figure 56: Graph of miles per gallon as a function of displacement

```
par(mar = c(5,4,4,4.5))
with(mtcars, {
  plot(disp, mpg, type='n',axes=F, ylim = c(10,35), xlim = c(50,500),
       xlab = 'displacement', ylab = '' )
  points(disp[cyl==4], mpg[cyl==4], pch=16, col=2)
  points(disp[cyl==6], mpg[cyl==6], pch=17, col=3)
  points(disp[cyl==8], mpg[cyl==8], pch=18, col=4)
  axis(1)
  axis(4)
  mtext('mpg',4, line = 2)
  title('Fuel Consumption', 'Data from 1974')
  arrows(470, 17, 470, 12, code = 2, length = 0.15)
  text(485,20,'worst\nyield',font=3,adj=1)
  arrows(100,34, 150,34, code=1, length = 0.2)
  text(220, 34, 'best yield', font=2)
  leg.txt <- c('4 cyl.', '6 cyl.', '8 cyl.')
  legend(400,35, leg.txt,col=2:4, pch=16:18)
})
```

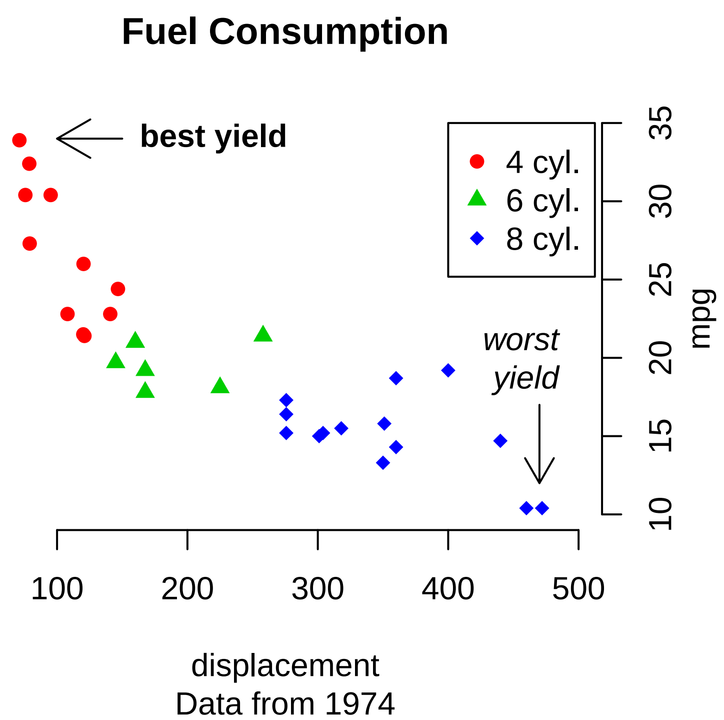


Figure 57: Graph of miles per gallon as a function of displacement with annotations

```r
par(oldpar)
```

14 Graphical Parameters

In addition to low-level commands, it is possible to modify the display of graphs changing the graphical parameters. These can be used as plotting function options (but this doesn't always work) or with the function `par` to permanently change the parameters, i.e. the graphs that are made next will also use the new parameters.

There are 73 graphical parameters. The complete list can be viewed using the `?par` instruction. We will not review them in detail and will only give some examples. You should at the help for `par` to have more information about the many features that can be controlled. The command

```r
par()
```

Gives a list of the current values for all the parameters, while

```r
old.par <- par(no.readonly = TRUE)
par(bg=7, bty='u', cex=1.5, col='blue', col.axis=4,
    font=2,lty='dashed', lwd=3, pch=3, las=2, tck =1)
```

stores in `old.par` the current values for the parameters that can be set by the user (some parameters can only be read but not changed). This will allow us to reset the parameters later.

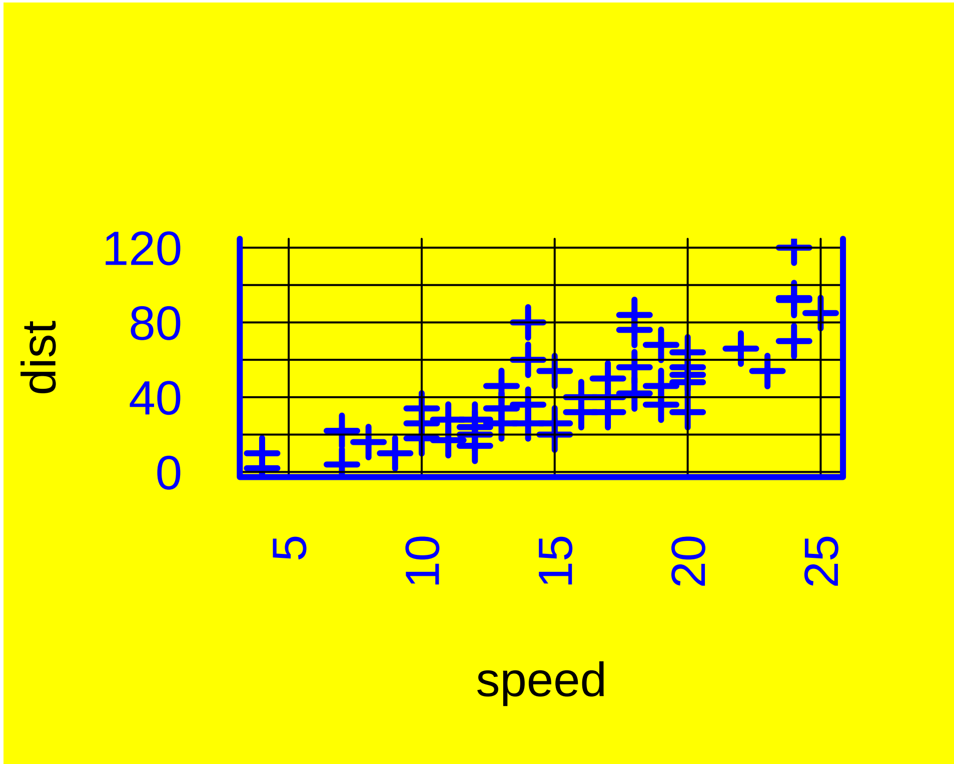


Figure 58: Graph of distance as a function of speed after changing several graphical parameters

Observe that any plot that is drawn after making the cahnges and before resetting the parameters to their default values will also have the same characteristics. For instance,

```
plot(iris)
```

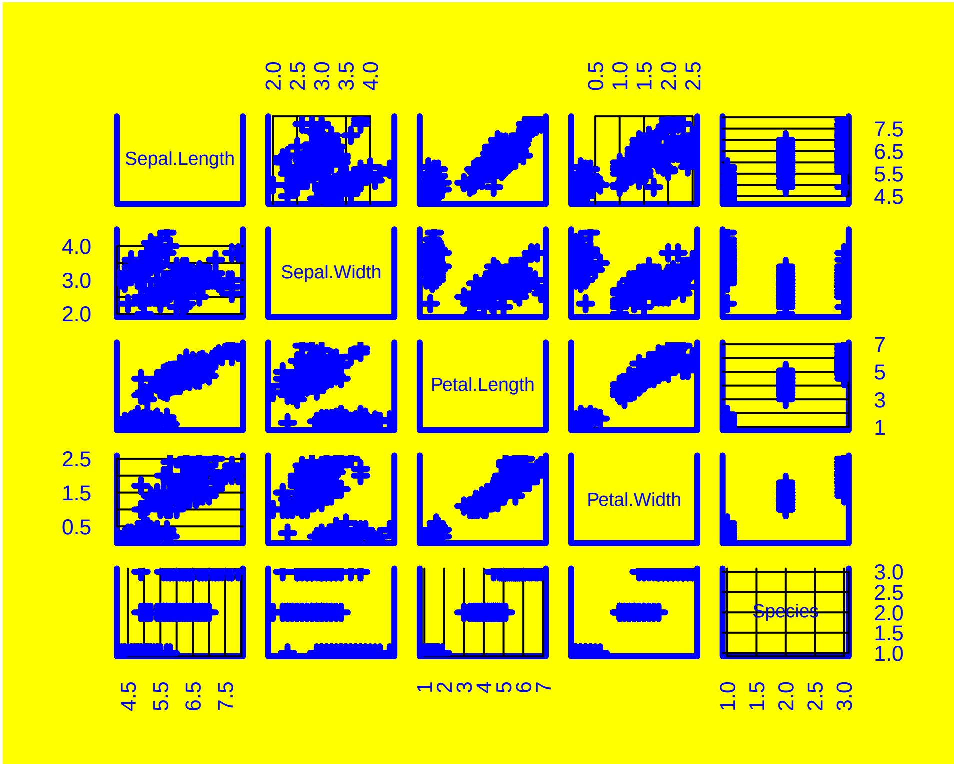


Figure 59: Graph of the dataset iris before restoring default values for the graphical parameters

We now reset the parameters and plot `cars` again:

```
par(old.par); plot(iris)
```

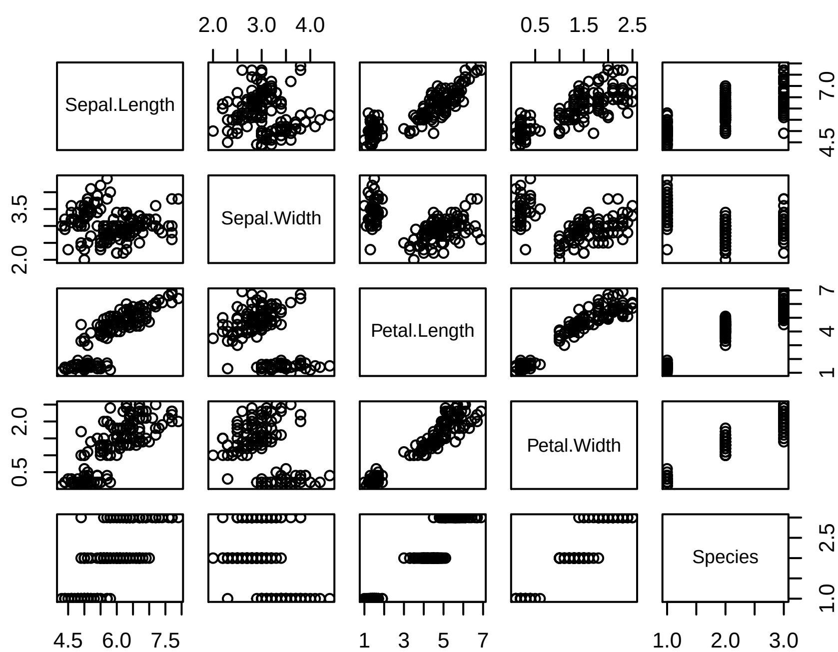


Figure 60: Graph of the dataset iris after restoring default values for the graphical parameters

15 Graphical Windows

There are a few ways to split a graphical window to display several graphs simultaneously. One possibility is to modify the graphical parameters using the function `par()` with the arguments `mfrow` or `mfcol`. This is probably the easiest way to split a graphical window. The argument for these functions is a vector with two components indicating the number of rows and columns, respectively. The difference between the functions is that the figures will be drawn by column in `mfcol` and by row in `mfrow`.

Another possibility is to use the `split.screen(c(m,n))` instruction that divides the window into m rows and n columns. Panels can be selected with `screen(r)` for $1 \leq r \leq m \cdot n$. `erase.screen()` deletes the last graph.

These functions are incompatible with functions such as `coplot` or `layout` and should not be used with multiple graphics devices.

```
split.screen(c(2, 1))          # split display into two screens
```

```
## [1] 1 2
```

```
split.screen(c(1, 2), 2)       # split bottom half in two
```

```
## [1] 3 4
```

```
plot(1:10)                     # screen 3 is active, draw plot
erase.screen()                 # forgot label, erase and redraw
plot(1:10, ylab = "ylab 3")
screen(1)                      # prepare screen 1 for output
plot(1:10)
screen(4)                      # prepare screen 4 for output
plot(1:10, ylab = "ylab 4")
screen(1, FALSE)               # return to screen 1, but do not clear
plot(10:1, axes = FALSE,
     lty = 2, ylab = "")       # overlay second plot
axis(4)                        # add tic marks to right-hand axis
title("Plot 1")
```

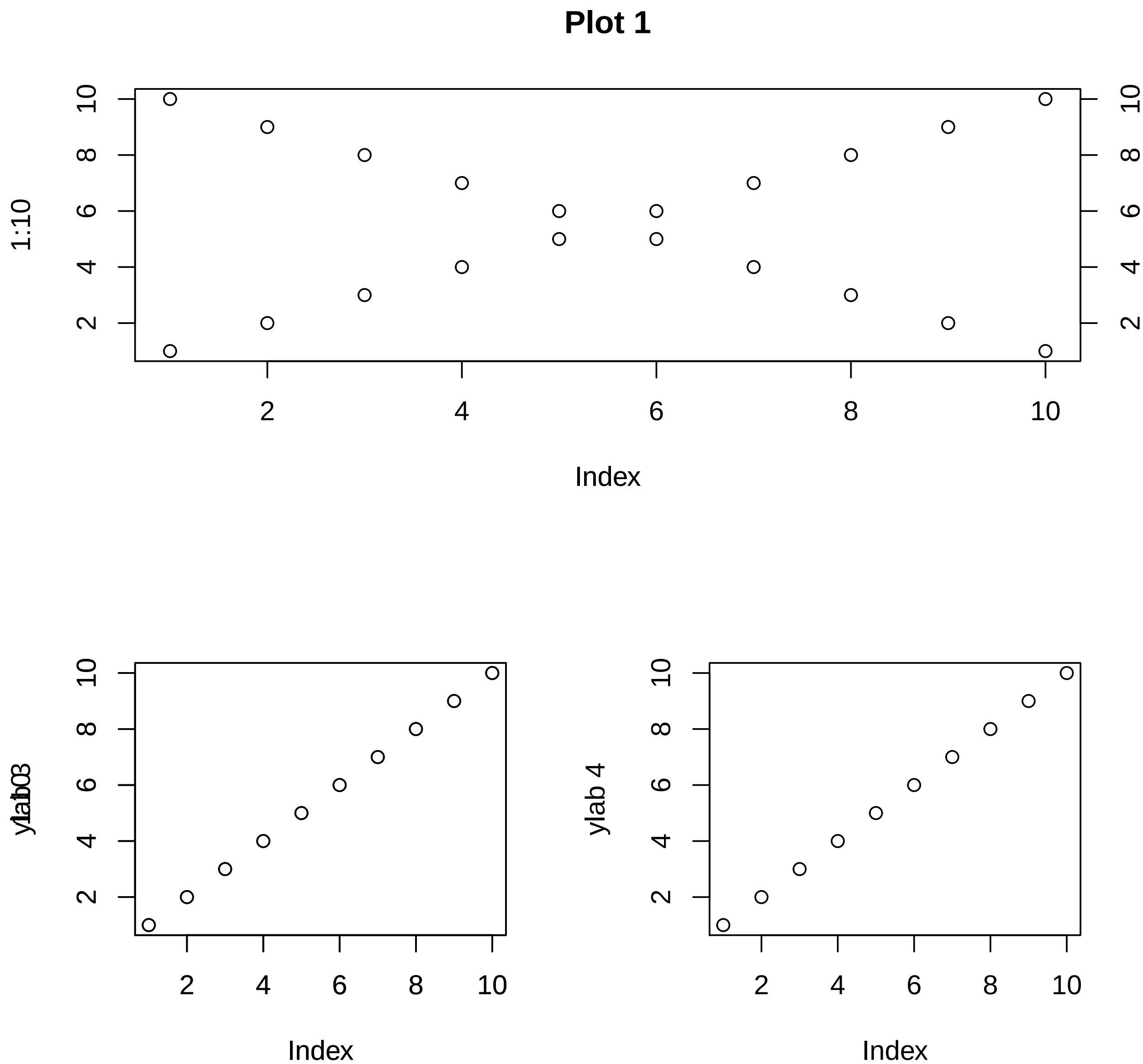


Figure 61: Example of the use of 'split.screen()'

```
close.screen(all = TRUE)     # exit split-screen mode
plot(1:10)
```

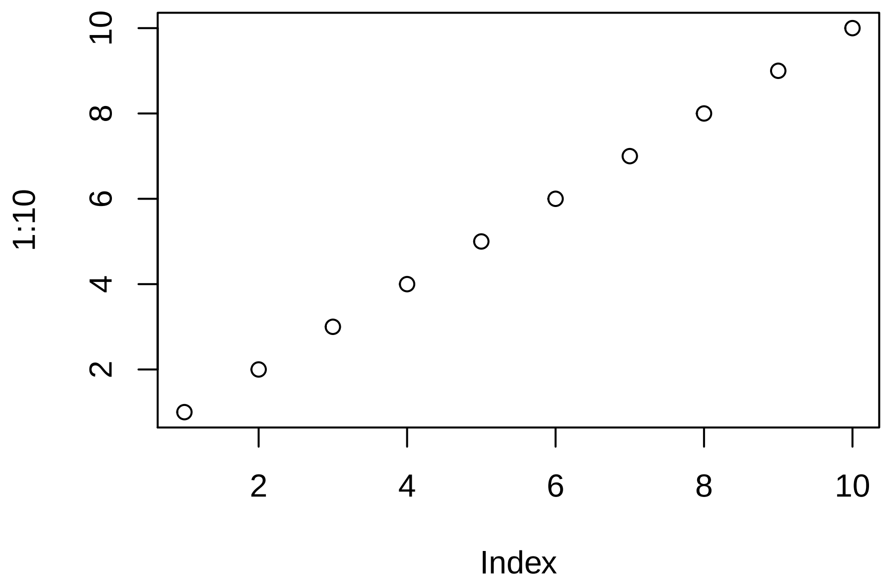


Figure 62: Graph after closing the split screen. Parameters are back to normal

15.1 `layout`

Another function that allows you to divide the graphic window is `layout`, which divides it into several panels in which the graphics will be drawn successively. The argument is a matrix of integers that indicates the number of divisions. For example, to divide the device into four parts we can use

```
(mat1 <- matrix(1:4,2,2))
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
layout(mat1)
layout.show(4)
```

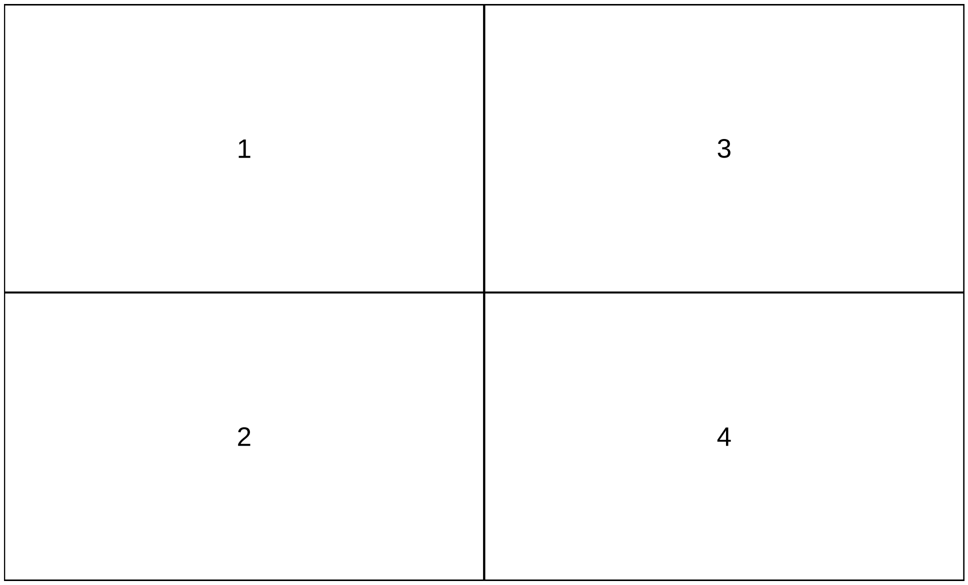


Figure 63: Example of the use of layout()

The following examples show some of the possibilities

```
layout(matrix(1:6,3,2))
layout.show(6)
layout(matrix(1:6,2,3))
layout.show(6)
layout(matrix(1:6,3,2,byrow=TRUE)) > layout.show(6)
```

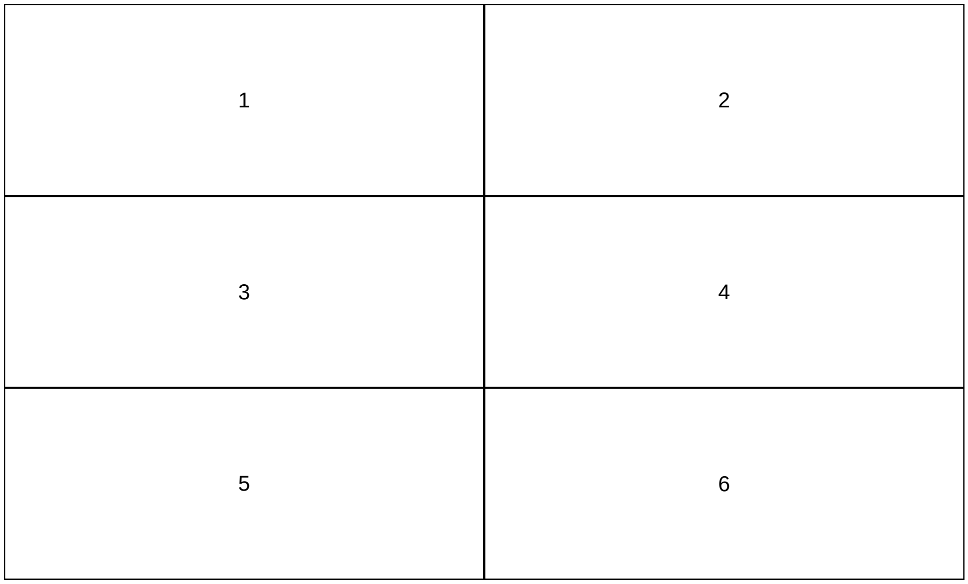


Figure 64: Example of the use of layout()

```
## logical(0)
```

```
(m <- matrix(c(1:3,3), 2, 2))
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    3
```

```r
layout(m)
layout.show(3)
```

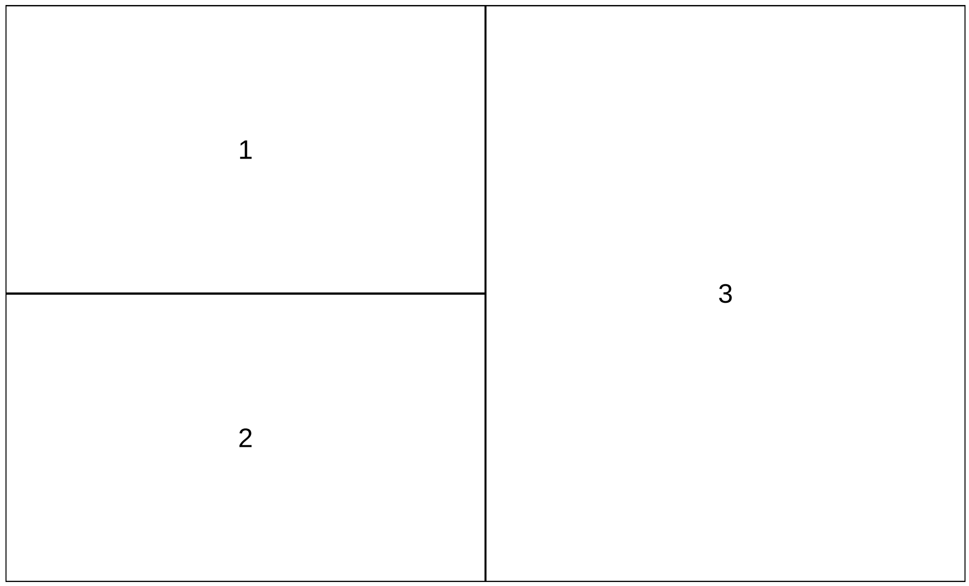


Figure 65: Example of the use of layout()

By default `layout()` divides the windows in equal-sized panels, but this can be modified with the options `widths` and `heights`

```r
m <- matrix(1:4, 2, 2)
layout(m, widths=c(1,3), heights=c(3,1))
layout.show(4)
```

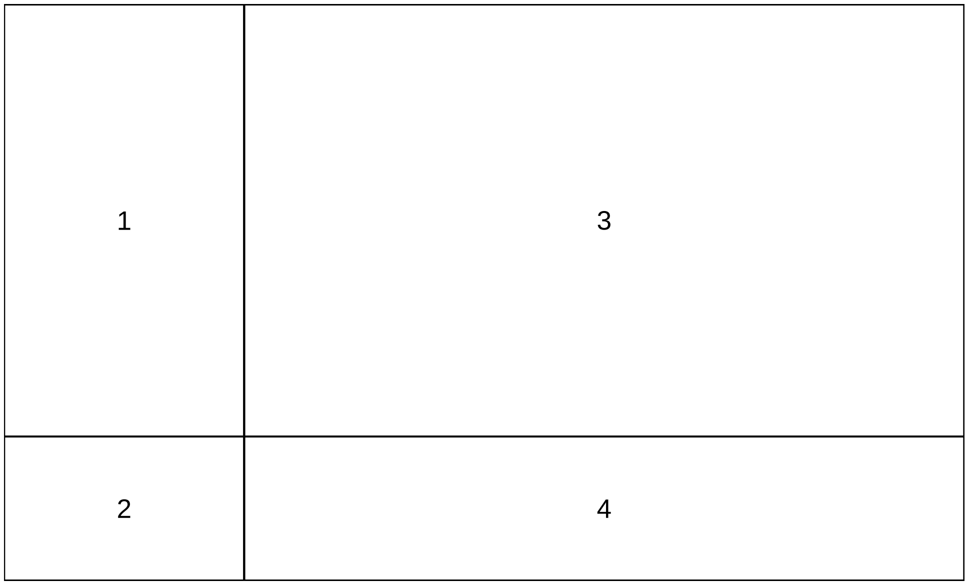


Figure 66: Example of the use of layout()

```
m <- matrix(c(1,1,2,1), 2, 2)
layout(m, widths=c(2,1), heights=c(1,2))
layout.show(2)
```

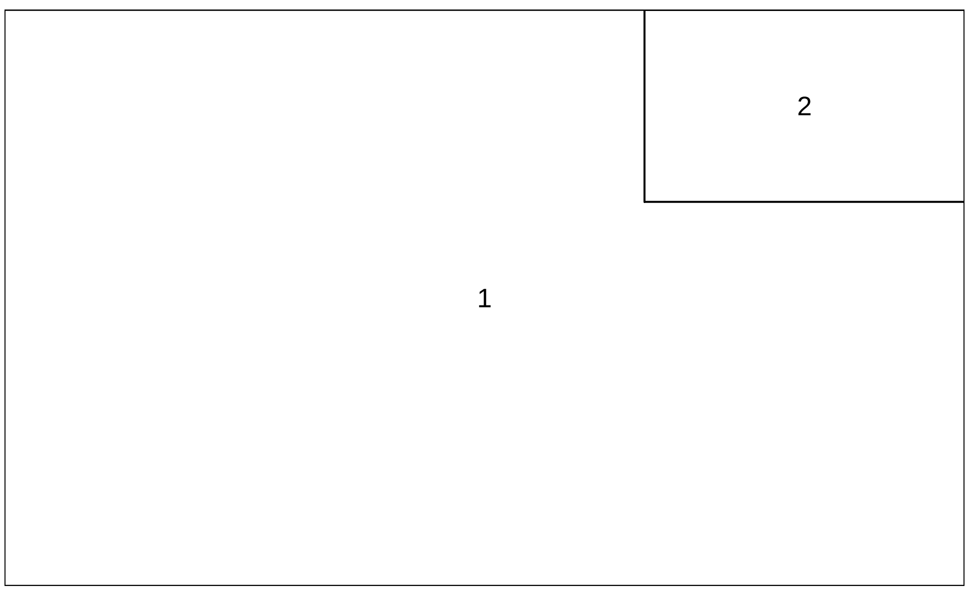


Figure 67: Example of the use of layout()

Matrix entries may take the value 0, allowing more complex layouts:

```
m <- matrix(0:3, 2, 2)
layout(m, widths=c(1,3), heights=c(1,3))
layout.show(3)
```

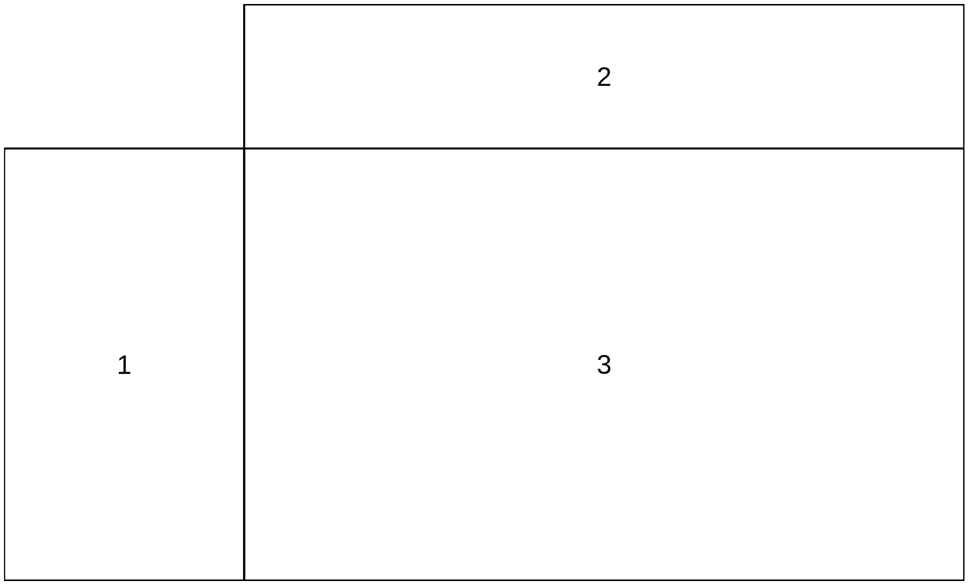


Figure 68: Example of the use of layout()

```
layout(m, widths=c(1,3), heights=c(1,3))
nf <- layout(matrix(c(2,0,1,3),2,2,byrow=TRUE), c(3,1), c(1,3), TRUE)
layout.show(nf)
```

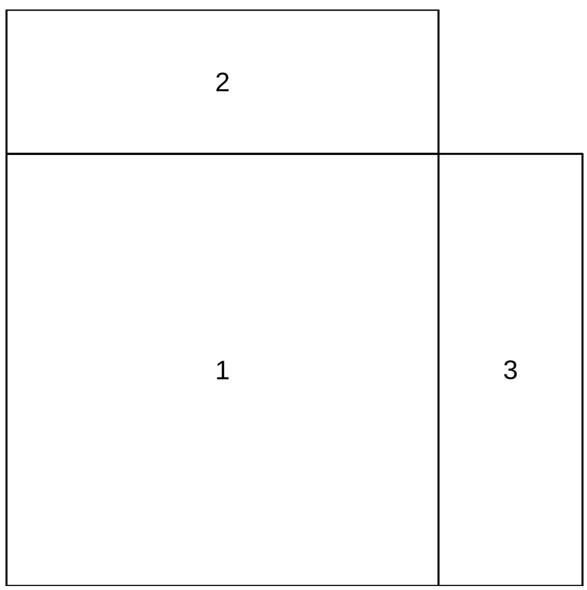


Figure 69: Example of the use of layout()

```
nf <- layout(matrix(c(2,0,1,3),2,2,byrow=TRUE), c(3,1), c(1,3), TRUE)
par(mar=c(2,2,2,2))
 x <- pmin(3, pmax(-3, rnorm(50)))
 y <- pmin(3, pmax(-3, rnorm(50)))
 xrange <- c(-3,3)
 yrange <- c(-3,3)
```

```
 xhist <- hist(x,breaks=seq(-3,3,0.5),plot=FALSE)
 yhist <- hist(y,breaks=seq(-3,3,0.5),plot=FALSE)
 top <- max(c(xhist$counts, yhist$counts))
plot(x, y, xlim=xrange, ylim=yrange, xlab='', ylab='')
barplot(xhist$counts, axes=FALSE, ylim=c(0, top), space=0)
barplot(yhist$counts, axes=FALSE, xlim=c(0, top), space=0, horiz=TRUE)
```

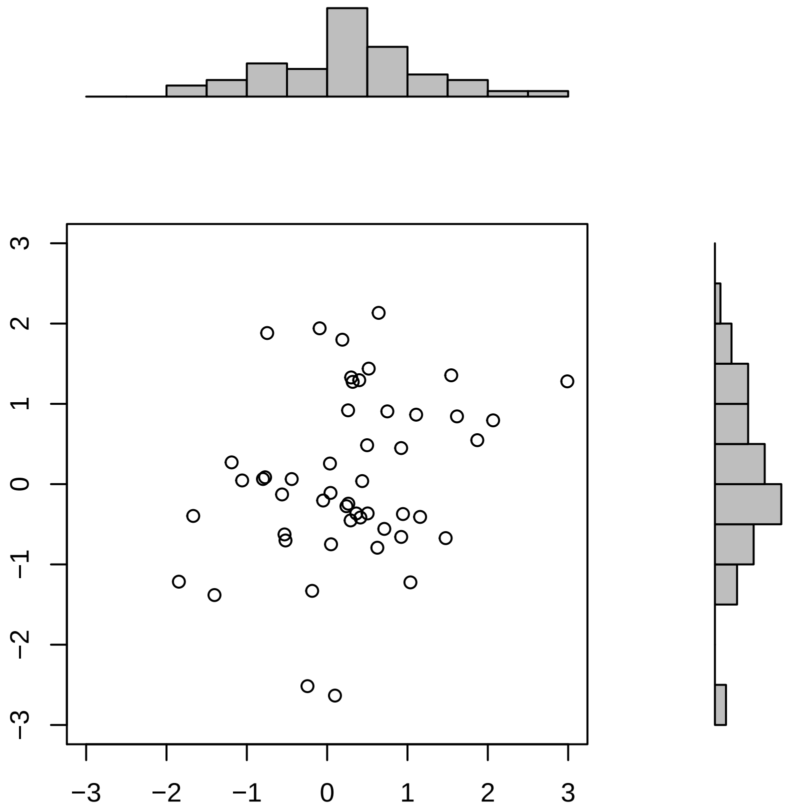


Figure 70: Example of the use of layout()

```
par(old.par)
```

16 Interactive functions

By the nature of these functions, their effect cannot be shown in these notes. The readers should try the commands by themselves.

16.1 `locator`

This function allows the user to click within a graph and obtain the coordinates of the selected point. It is also possible to use it to place symbols where you click or to draw segments between the selected points. The syntax is

```
locator(n,type)
```

With this instruction `R` expects the user to select n points on the active graph. The argument `type` allows to draw on the selected points and has the same syntax as for high level graphical commands. The default option is not to draw anything.

The `locator()` output are the coordinates of the points selected as a list with two components.

```r
plot(cars)
locator(3,type='n')
locator(2,type='l')
text(locator(1), 'Punto', adj=0)
```

16.2 `identify`

This function can be used to identify data in a graph. The data closest to the pointer position is identified by clicking.

```r
identify(x, y, labels).
```

The procedure is similar to locator but instead of identifying the coordinates, the point is identified through labels. If labels are not present in the function call, the row in which the data is in the matrix is used as label. To finish the identification process press the right mouse button and select Stop.