Introduction to R part 2

Instructor: Christian Capezza
Course leader: Prof. Biagio Palumbo
Course: Statistics for Technology, a.y. 2019/2020
MSc in Mechanical Engineering for Design and Production

25 October 2019

Control flows

if

```
if (5 > 3) aa <- 1
if (5 < 3) bb <- 1
```

if/else

```
if (5 > 3) cc <- 2 else cc <- 3
if (5 < 3) dd <- 2 else dd <- 3
```

if more than one instruction, use curly brackets `{}`

```
if (5 > 3) {
  ee <- "a"
  ff <- "b"
  } else {
    gg <- "c"
    hh <- "d"
  }
```

Vectorized if/else with the function `ifelse`

```
vec1 <- c(2, 5, 6, - 8.7)
vec2 <- c(5.9, - 1, 56, 0)
ifelse(vec1 > vec2, 10, - 7)
```

```
## [1] -7 10 -7 -7
```

`for` loop

```
for (ii in 1:3) {
  print(ii)
}
```

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

```
vec <- 1:10
for (ii in 1:length(vec)) {
  print(ii)
}
```

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

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Note: it's better to use seq_along

```
1:length(vec)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq_along(vec) ## equivalent
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
for (ii in seq_along(vec)) {
  print(ii)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Why? suppose `vec` has length 0, as result of some filtering

```
vec <- which(c(1, 3, 4) < 0) # the result is empty vector
```

the following returns error, try!

```
# for (ii in 1:seq_along(vec)) {
#   print(ii)
# }
```

then use

```
for (ii in seq_along(vec)) { #returns nothing
  print(ii)
}
```

while

```
ii <- 0
while(ii < 100) {
  print(ii)
  ii <- (ii + 1) ^ 2
}
```

```
## [1] 0
## [1] 1
## [1] 4
## [1] 25
```

Functions

Each function has the following structure

- Name: a function is a class like numeric, factor, character, etc. then it can be saved to a variable. Sometimes we can use the function "on the fly", i.e. we can avoid to save the function to a variable. You will understand with the apply functions
- Input(s)
- Code: the code processes the input variables and must provide something at the end
- Output: one single output can be returned. If more than one is desired, use a list. Usually the last statement produces the output. `return()` can be omitted

Note: when we define a function, we are not executing anything yet! We must call the function later

Example:

```
f <- function(x) {
  return(x + 10) ## return can be omitted, see below
}
f <- function(x) {
  x + 10
}
f <- function(x) x + 10 # if one statement only, you can omit {}
```

Here:

- Name: `f`, we are saving the function object to the variable `f`
- Input: `x`
- Code: add 10 to the input `x`
- Output: return `x` + 10

What happens if we call `f`?

```
f
```

```
## function(x) x + 10
```

Now let us use `f`. We can insert the inputs in parenthesis by calling them by name followed by =, or insert the input directly (you must respect the order estabilished when defining the function)

```
f(x = 1.6)
```

```
## [1] 11.6
```

```
f(1.6)
```

```
## [1] 11.6
```

We can save the output to a variable

```
aa <- f(- 7)
aa
```

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

Another example. More inputs, more outputs, more instructions

```
f <- function(x, y) {
  z <- x ^ 2 * y
  sin(z)
}
f(x = 1, y = 3)
```

```
## [1] 0.14112
```

```
f(1, 3) #equivalent, you must know the first argument is x and the second is y
```

```
## [1] 0.14112
```

```
# f(x = 1) produces error, all arguments must be provided
f(y = 4, x = 9) # if you specify names of input arguments, you can change the order
```

```
## [1] -0.4040652
```

You can specify a default value for some inputs when defining a function. In this case, you can omit the corresponding input

```
f <- function(x, y = 2) {
  z <- x ^ 2 * y
  sin(z)
}
f(1) # here y takes the default value 2
```

```
## [1] 0.9092974
```

```
f(1, 3) # you can overwrite the y value of course
```

```
## [1] 0.14112
```

Environments (very basics)

When calling a function, a function uses its internal environment. It means it does not overwrite values in the global environment

```
yy <- 2
fun <- function(xx) {
  yy <- 1
  xx + yy
}
fun(3)
```

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

```
yy ## yy is not overwritten by the function call
```

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

Instead, if you use some variables in a function not defined in the function body and not available among the function inputs, R looks in the environment level above. If possible, avoid this and always try to write functions tha only uses variables locally defined and inputs

```
yy <- 3
fun <- function(xx) {
  xx + yy
}
fun(4)
```

```
## [1] 7
```

Apply functions

The apply functions apply a function to all the elements of an object (generally, a vector, list, data frame or matrix). For all these apply functions, there is a function that we want to apply to these elements.

apply

Apply a function to all rows or columns of a **matrix**.

Suppose we want to calculate the sum of squares of each row of a matrix `M`. Three arguments:

- the matrix objects whose rows/columns we want to iterate on
- 1 if we want to iterate over rows, 2 if columns
- the function we want to apply to each row (or column)

```
fun <- function(x) sum(x ^ 2)
M <- cbind(c(1, 2, 3), c(4, 5, 6))
M
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
apply(M, 1, fun)
```

```
## [1] 17 29 45
```

```
apply(M, 2, fun)
```

```
## [1] 14 77
```

Note that often we need to use functions like `fun` only in this circumstance. We don't mean to use it elsewhere. In this case, we can use an **anonymous function**, i.e. we avoid to assign it to a variable and use it directly. This holds for all the apply functions:

```
apply(M, 1, function(x) sum(x ^ 2))
```

```
## [1] 17 29 45
```

lapply

It works with vectors/lists. Two arguments:

- the vector/list we want to iterate on
- the function we want to apply to each element

Example: we want the length of all the elements of a list

```
ll <- list(num = 1:10, cha = c("a", "b"))
lapply(ll, function(x) length(x)) ## here it is useless to use anonymous function!
```

```
## $num
## [1] 10
##
## $cha
## [1] 2
```

```
lapply(ll, length)
```

```
## $num
## [1] 10
```

```
##
## $cha
## [1] 2
```

```
vv <- c(1, 2, 3)
lapply(vv, function(x) x ^ 2)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
```

sapply

Like lapply, but instead of returning a list, if possible, it returns a vector/matrix

```
ll <- list(num = 1:10, cha = c("a", "b"))
sapply(ll, length)
```

```
## num cha
##  10   2
```

tapply

Essentially, the meaning is to split observations in subgroups according to factor variables, and to apply the same function to each subgroup. Let us consider a dataset with more than one factor variable

```
heart <- read.csv("data/Heart.csv")
head(heart)
```

```
##   X Age Sex    ChestPain RestBP Chol Fbs RestECG MaxHR ExAng Oldpeak Slope
## 1 1  63   1      typical    145  233   1       2   150     0     2.3     3
## 2 2  67   1 asymptomatic    160  286   0       2   108     1     1.5     2
## 3 3  67   1 asymptomatic    120  229   0       2   129     1     2.6     2
## 4 4  37   1   nonanginal    130  250   0       0   187     0     3.5     3
## 5 5  41   0   nontypical    130  204   0       2   172     0     1.4     1
## 6 6  56   1   nontypical    120  236   0       0   178     0     0.8     1
##   Ca       Thal AHD
## 1  0      fixed  No
## 2  3     normal Yes
## 3  2 reversable Yes
## 4  0     normal  No
## 5  0     normal  No
## 6  0     normal  No
```

```
str(heart)
```

```
## 'data.frame':    303 obs. of  15 variables:
##  $ X        : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Age      : int  63 67 67 37 41 56 62 57 63 53 ...
##  $ Sex      : int  1 1 1 1 0 1 0 0 1 1 ...
##  $ ChestPain: Factor w/ 4 levels "asymptomatic",..: 4 1 1 2 3 3 1 1 1 1 ...
##  $ RestBP   : int  145 160 120 130 130 120 140 120 130 140 ...
##  $ Chol     : int  233 286 229 250 204 236 268 354 254 203 ...
```

```
##  $ Fbs      : int  1 0 0 0 0 0 0 0 0 1 ...
##  $ RestECG  : int  2 2 2 0 2 0 2 0 2 2 ...
##  $ MaxHR    : int  150 108 129 187 172 178 160 163 147 155 ...
##  $ ExAng    : int  0 1 1 0 0 0 0 1 0 1 ...
##  $ Oldpeak  : num  2.3 1.5 2.6 3.5 1.4 0.8 3.6 0.6 1.4 3.1 ...
##  $ Slope    : int  3 2 2 3 1 1 3 1 2 3 ...
##  $ Ca       : int  0 3 2 0 0 0 2 0 1 0 ...
##  $ Thal     : Factor w/ 3 levels "fixed","normal",..: 1 2 3 2 2 2 2 2 3 3 ...
##  $ AHD      : Factor w/ 2 levels "No","Yes": 1 2 2 1 1 1 2 1 2 2 ...
```

```
summary(heart)
```

```
##        X              Age             Sex               ChestPain
##  Min.   :  1.0   Min.   :29.00   Min.   :0.0000   asymptomatic:144
##  1st Qu.: 76.5   1st Qu.:48.00   1st Qu.:0.0000   nonanginal  : 86
##  Median :152.0   Median :56.00   Median :1.0000   nontypical  : 50
##  Mean   :152.0   Mean   :54.44   Mean   :0.6799   typical     : 23
##  3rd Qu.:227.5   3rd Qu.:61.00   3rd Qu.:1.0000
##  Max.   :303.0   Max.   :77.00   Max.   :1.0000
##
##      RestBP           Chol            Fbs            RestECG
##  Min.   : 94.0   Min.   :126.0   Min.   :0.0000   Min.   :0.0000
##  1st Qu.:120.0   1st Qu.:211.0   1st Qu.:0.0000   1st Qu.:0.0000
##  Median :130.0   Median :241.0   Median :0.0000   Median :1.0000
##  Mean   :131.7   Mean   :246.7   Mean   :0.1485   Mean   :0.9901
##  3rd Qu.:140.0   3rd Qu.:275.0   3rd Qu.:0.0000   3rd Qu.:2.0000
##  Max.   :200.0   Max.   :564.0   Max.   :1.0000   Max.   :2.0000
##
##      MaxHR           ExAng          Oldpeak         Slope
##  Min.   : 71.0   Min.   :0.0000   Min.   :0.00   Min.   :1.000
##  1st Qu.:133.5   1st Qu.:0.0000   1st Qu.:0.00   1st Qu.:1.000
##  Median :153.0   Median :0.0000   Median :0.80   Median :2.000
##  Mean   :149.6   Mean   :0.3267   Mean   :1.04   Mean   :1.601
##  3rd Qu.:166.0   3rd Qu.:1.0000   3rd Qu.:1.60   3rd Qu.:2.000
##  Max.   :202.0   Max.   :1.0000   Max.   :6.20   Max.   :3.000
##
##        Ca                Thal      AHD
##  Min.   :0.0000   fixed     : 18   No :164
##  1st Qu.:0.0000   normal    :166   Yes:139
##  Median :0.0000   reversable:117
##  Mean   :0.6722   NA's      :  2
##  3rd Qu.:1.0000
##  Max.   :3.0000
##  NA's   :4
```

Consider the factor Sex, we can partition people according to sex. `tapply` wants three arguments:

- a vector of n elements, we want to apply a function to each subgroup of this vector. the subgroup is identified by the second argument
- a factor of n elements, which we use to indicate the subgroup that each of the n elements of the first argument belongs to. If the grouping is according to more than one factor, use a list
- the function to apply to each subgroup

```
tapply(heart$Age, heart$Sex, mean)
```

```
##        0        1
```

```
## 55.72165 53.83495
```

```
tapply(heart$Age, list(heart$Sex, heart$AHD), mean)
```

```
##         No      Yes
## 0 54.55556 59.08000
## 1 51.04348 56.08772
```

When you just want to count the number of elements in each subgroup, just use table

```
tapply(heart$Age, heart$Sex, length)
```

```
##   0   1
##  97 206
```

```
table(heart$Sex) # equivalent
```

```
##
##   0   1
##  97 206
```

```
tapply(heart$Age, list(heart$Sex, heart$AHD), length)
```

```
##   No Yes
## 0 72  25
## 1 92 114
```

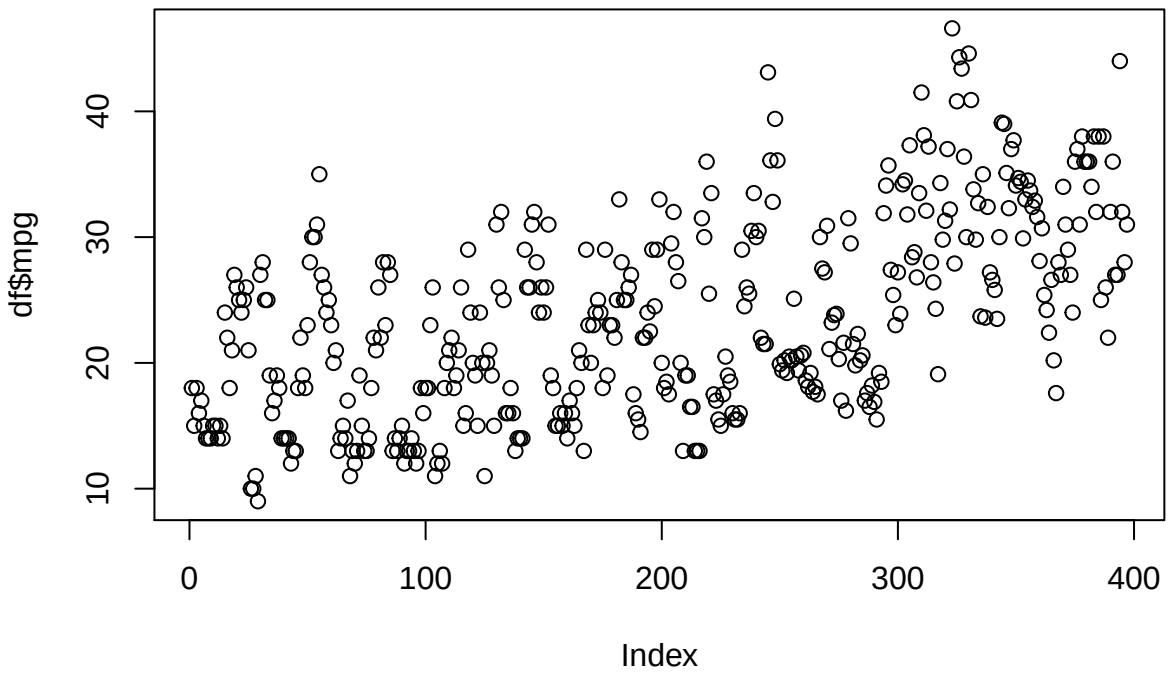
```
table(heart$Sex, heart$AHD) # equivalent
```

```
##
##      No Yes
##   0  72  25
##   1  92 114
```

Plots: the very basics

Graphics: one variable (numeric)

```
df <- read.csv("data/Auto.csv")
plot(df$mpg) # Plots n vector values against 1:n
```



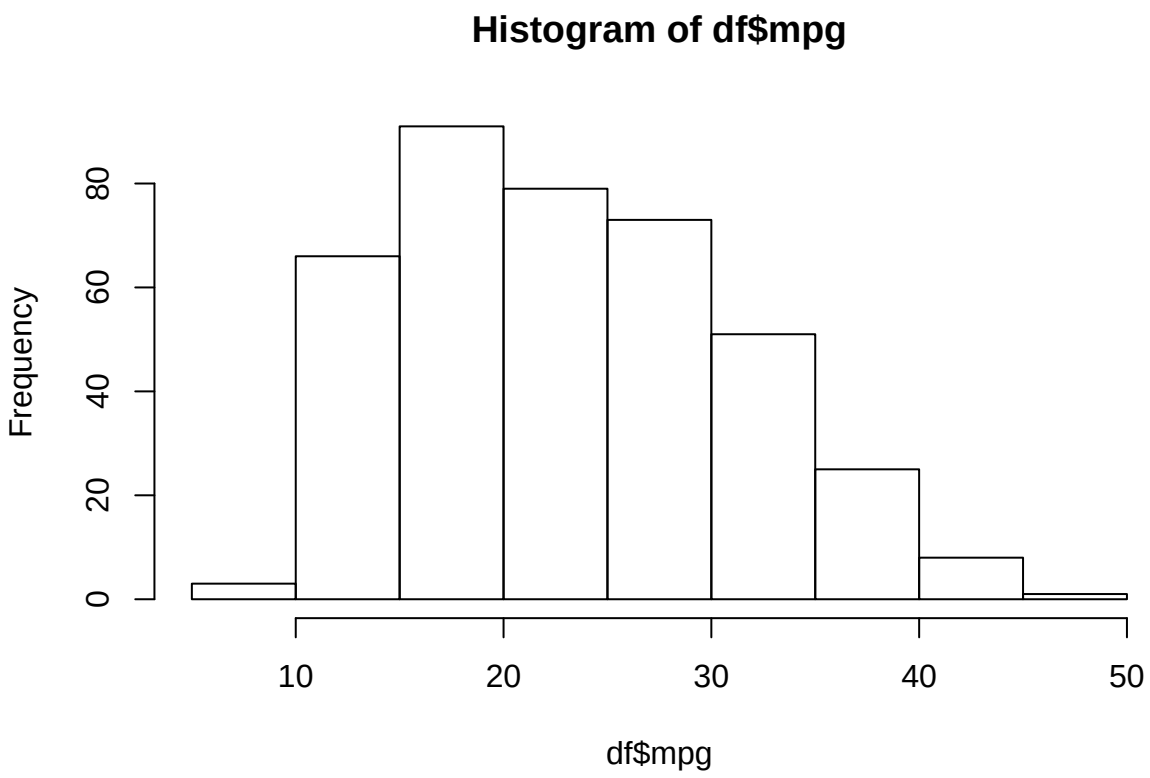
Graphics: one variable (numeric)

Results not shown: try them!

```
plot(df$mpg, type = "l") # Lines instead of points
plot(df$mpg, col = "red") # Set points/lines colours
plot(df$mpg, pch = 16) # Change point shape
plot(df$mpg, cex = 3) # Change point/line size
plot(df$mpg, type = "l", lty = 2) # Change line type
plot(df$mpg, main = "Title") # Set plot title
plot(df$mpg, xlab = "x axis") # Set x axis title
plot(df$mpg, ylab = "y axis") # Set y axis title
plot(df$mpg, xlim = c(10, 50)) # Set limits of x axis
plot(df$mpg, ylim = c(0, 60)) # Set limits of y axis
```

Graphics: one variable (numeric)

```
hist(df$mpg) # histogram of frequencies
```



Graphics: one variable (numeric)

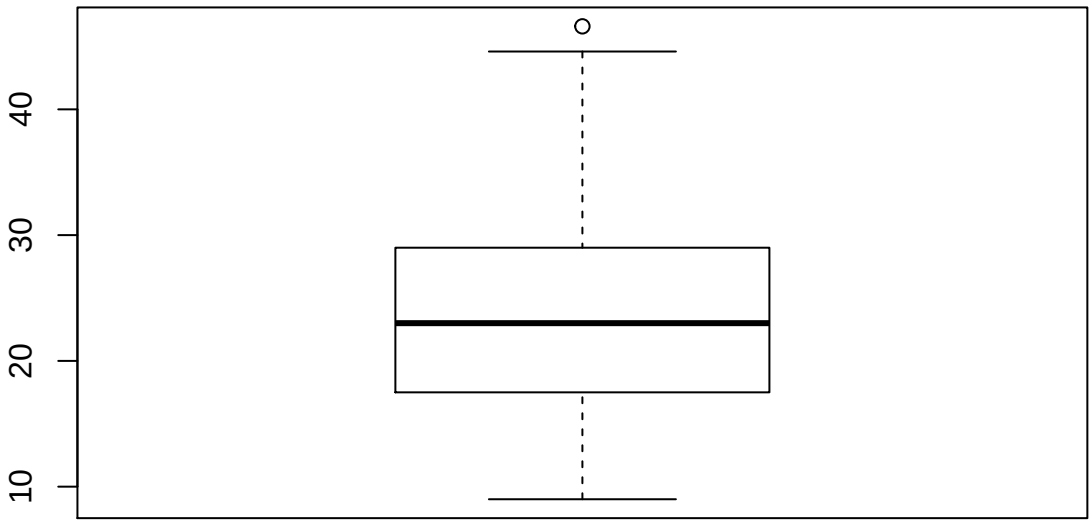
`probability = TRUE` is useful in cases you want to plot densities over histogram. Note: If you want to add a curve or points over an existing plot, after creating the first plot (e.g. `plot`, or `hist`), use `lines` or `points`.

Results not shown: try them!

```
hist(df$mpg, breaks = 40) # set number of bins
hist(df$mpg, probability = TRUE) # histogram of densities
lines(density(df$mpg)) # adds kernel density estimate
```

Graphics: one variable (numeric)

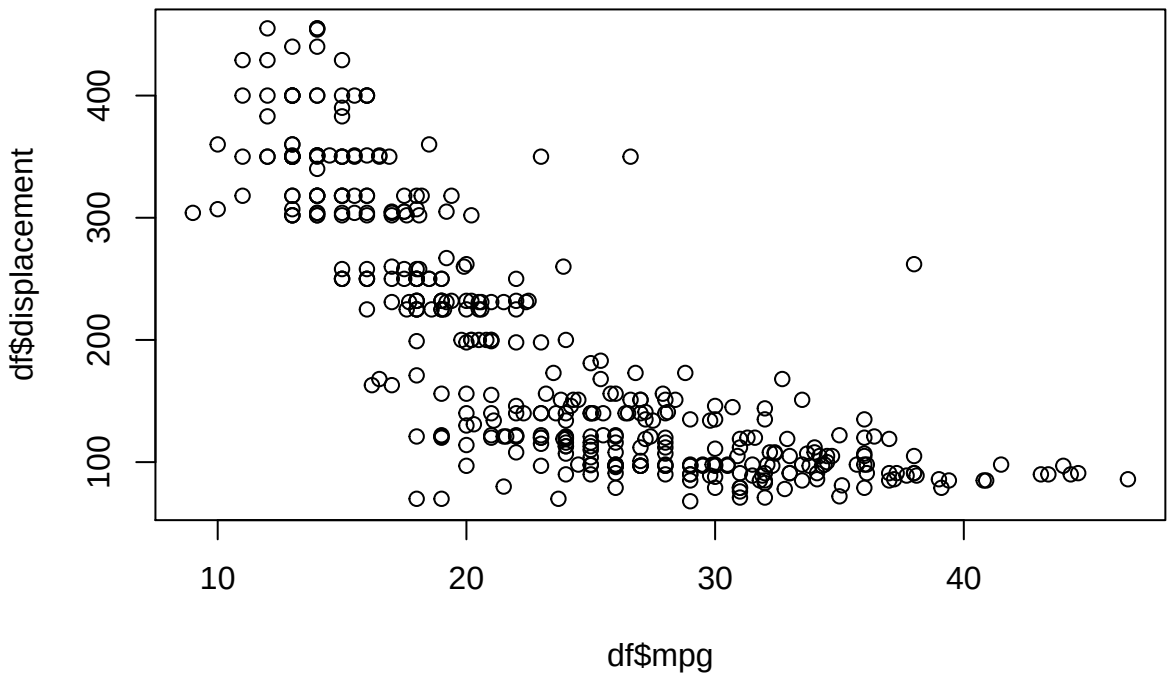
```
boxplot(df$mpg)
```



Graphics: two variables (numeric)

Scatterplot of two variables

```
plot(df$mpg, df$displacement)
```

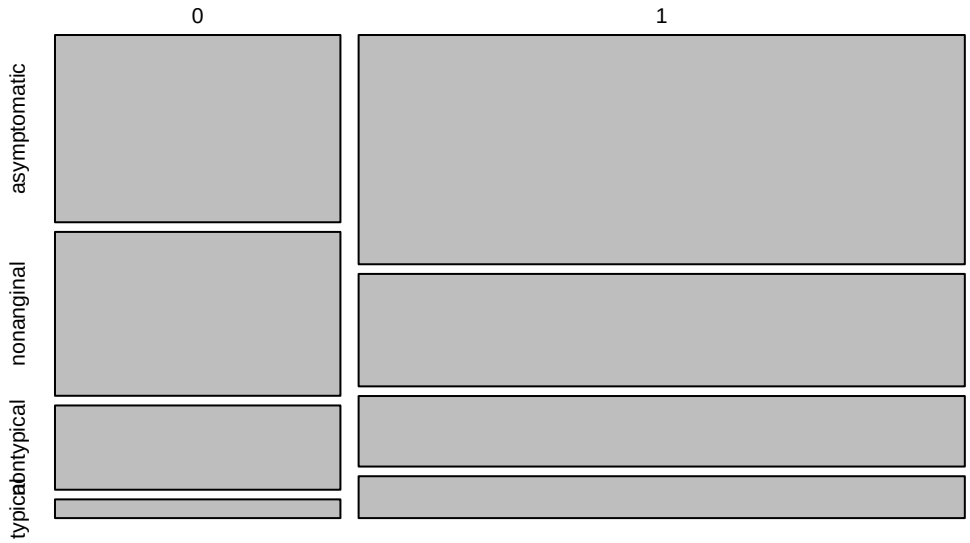


Graphics: one variable (factor)

```
plot(table(heart$Sex))
```

Graphics: two variables (factors)

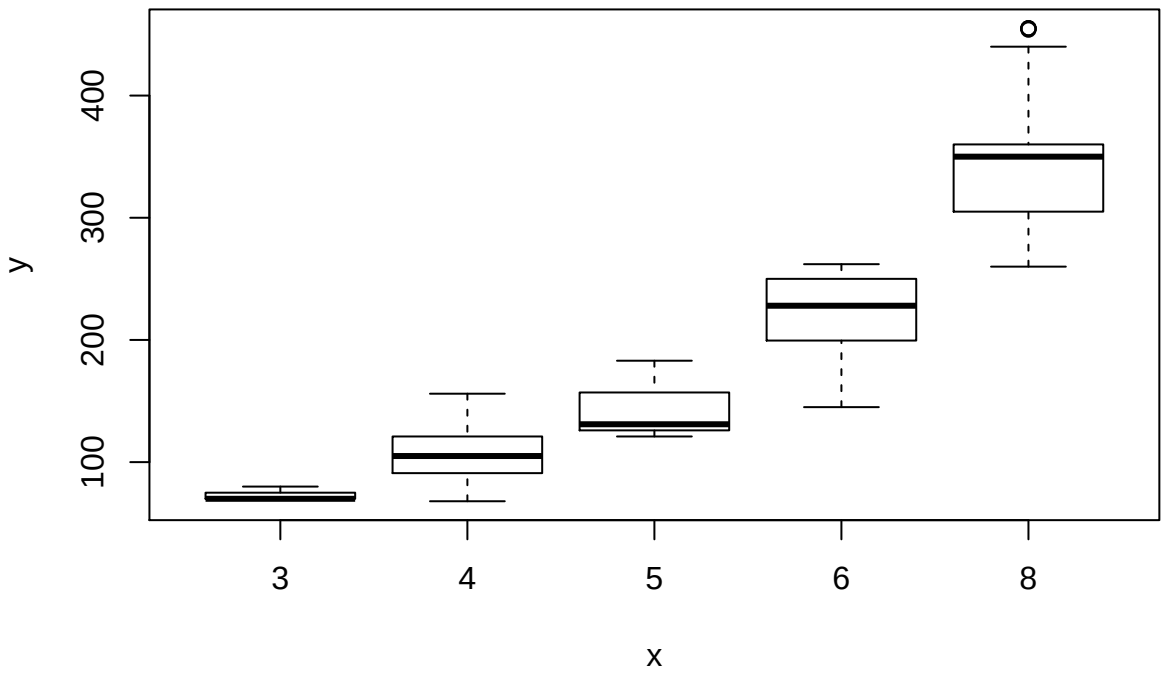
```
plot(table(heart$Sex, heart$ChestPain))
```



Graphics: two variables (one factor one numeric)

Each box plot is the one of all observations of displacement corresponding to the given level of the factor variable.

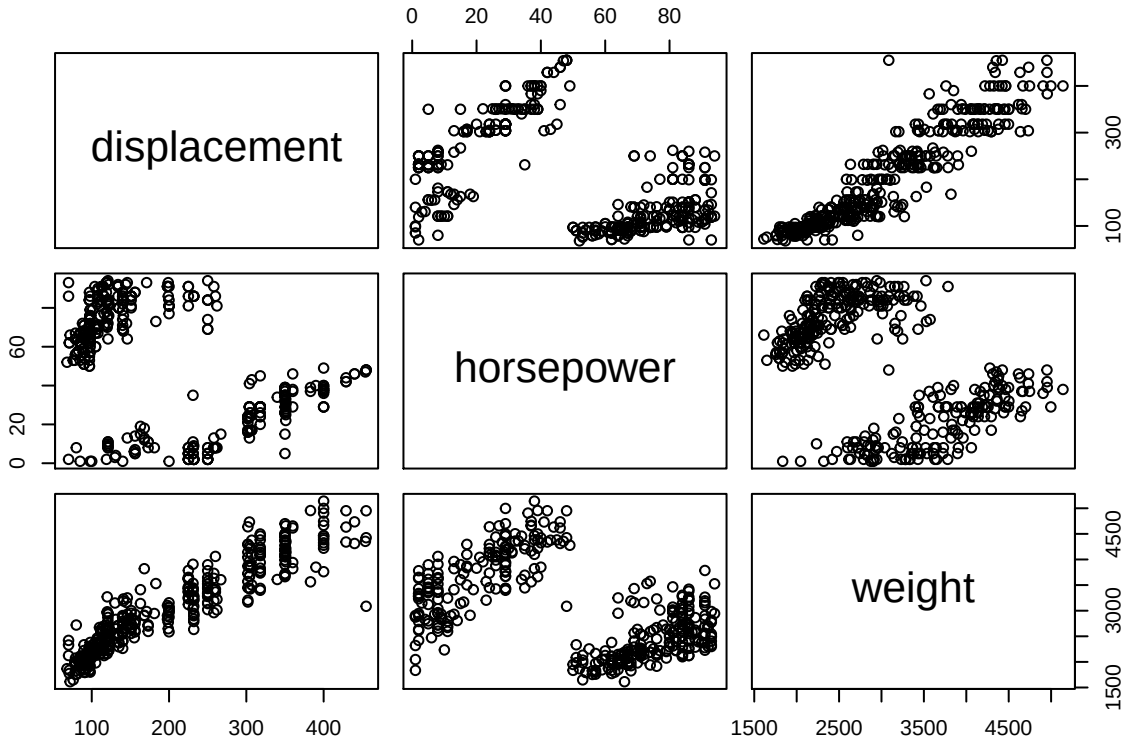
```
plot(factor(df$cylinders), df$displacement)
```



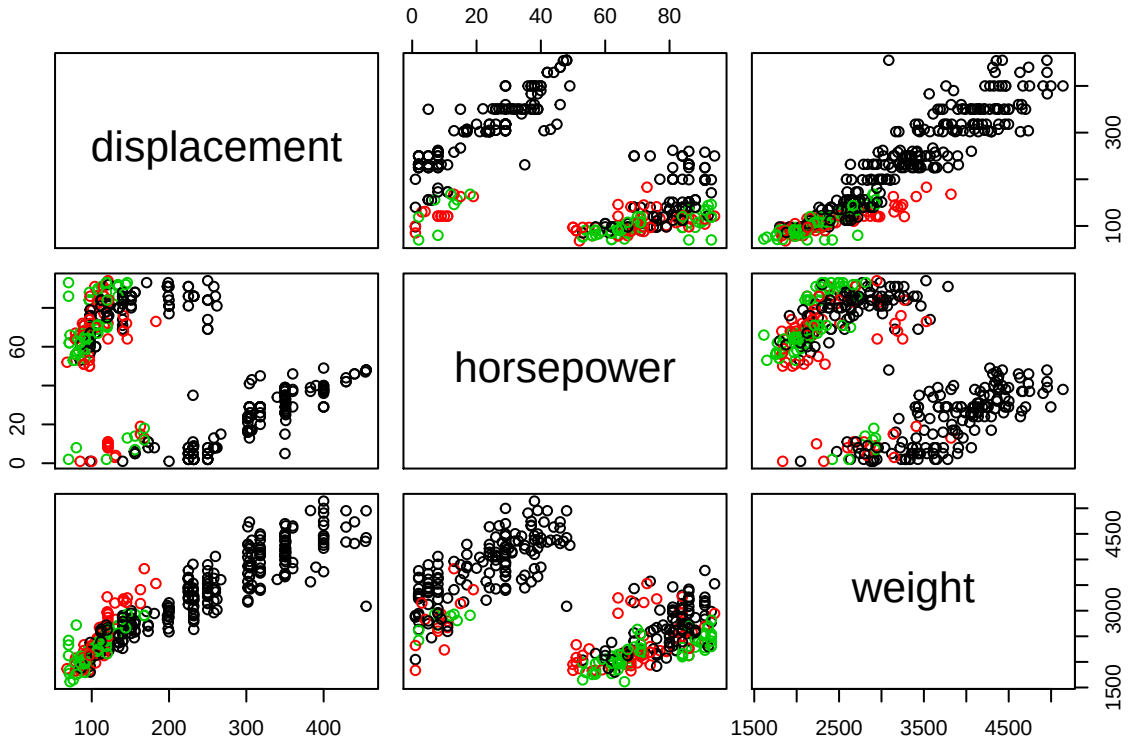
Graphics: more than two variables (numeric)

Scatterplot of three variables

```
pairs(df[, 3:5])
```

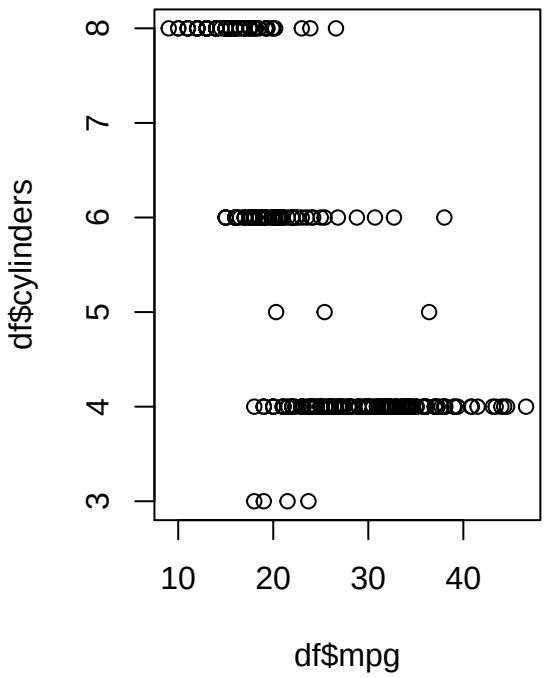


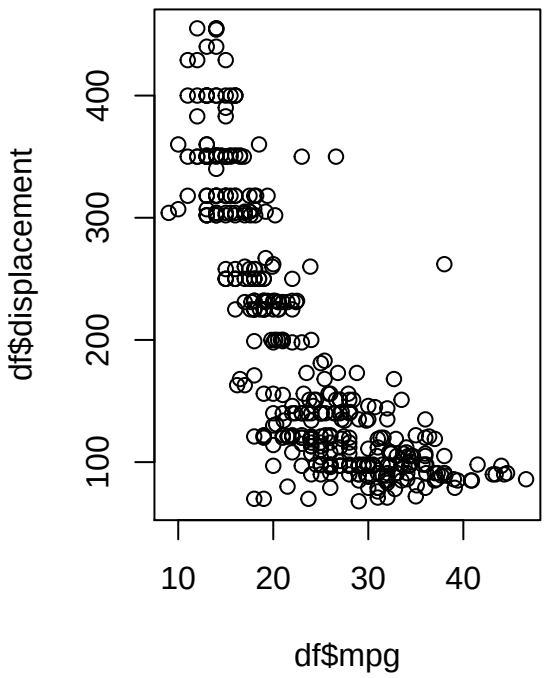

```
pairs(df[, 3:5], col = df$origin)
```



Multiple graphs

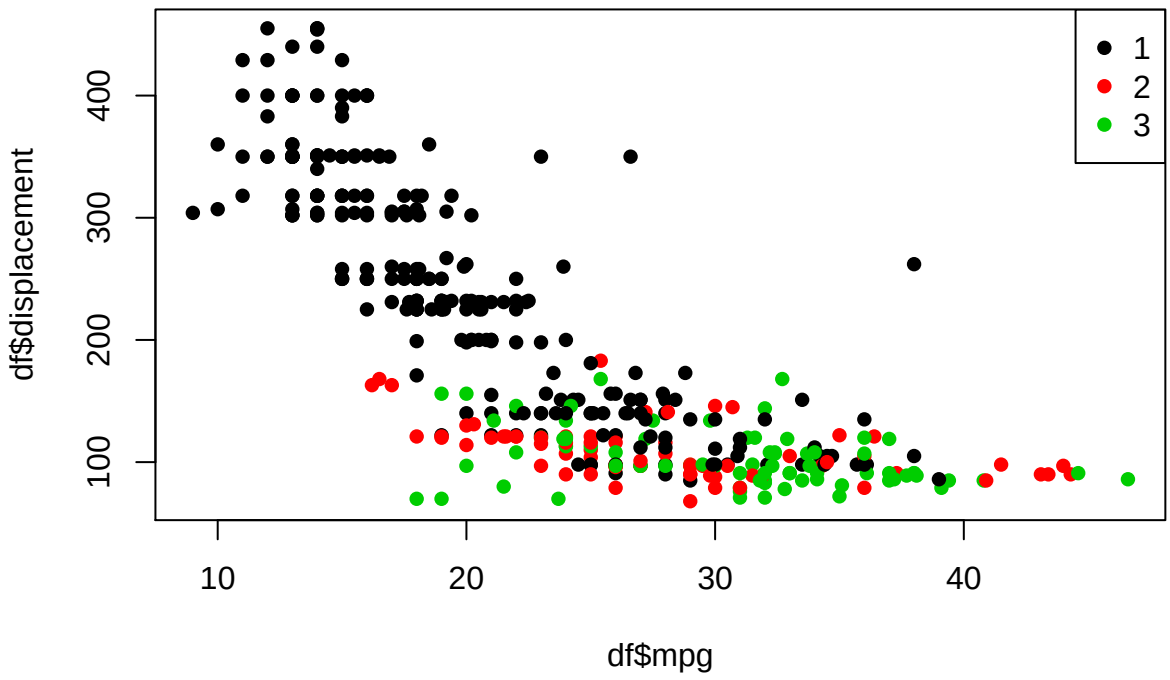
```
par(mfrow = c(1, 2))
plot(df$mpg, df$cylinders)
plot(df$mpg, df$displacement)
```





Add legend

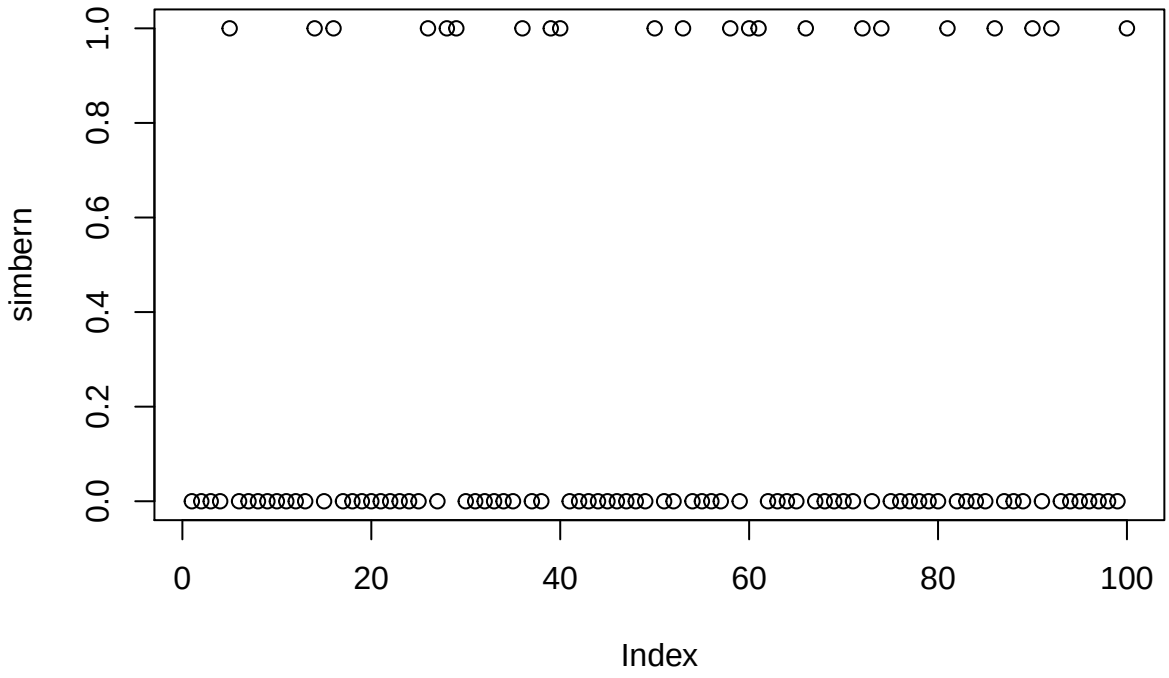
```
plot(df$mpg, df$displacement, col = factor(df$origin), pch = 16)
legend("topright", legend = levels(factor(df$origin)),
       col = levels(factor(df$origin)), pch = 16)
```



Some probability distributions

```
set.seed(1234)
nsim <- 1e2
probs <- c(0, .25, .5, .75, 1)

simbern <- rbinom(nsim, 1, .3)
plot(simbern)
```



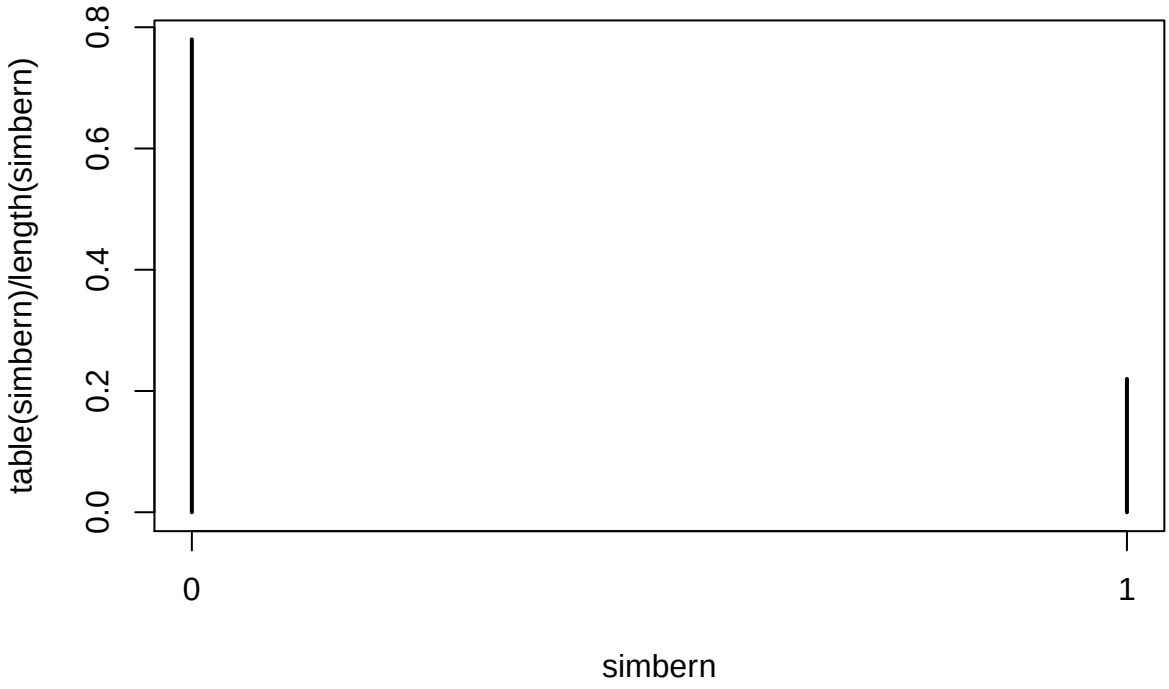

```
table(simbern)
```

```
## simbern
##  0  1
## 78 22
```

```
table(simbern) / length(simbern)
```

```
## simbern
##    0    1
## 0.78 0.22
```

```
plot(table(simbern) / length(simbern))
```




```
dbinom(0:1, 1, .3)
```

```
## [1] 0.7 0.3
```

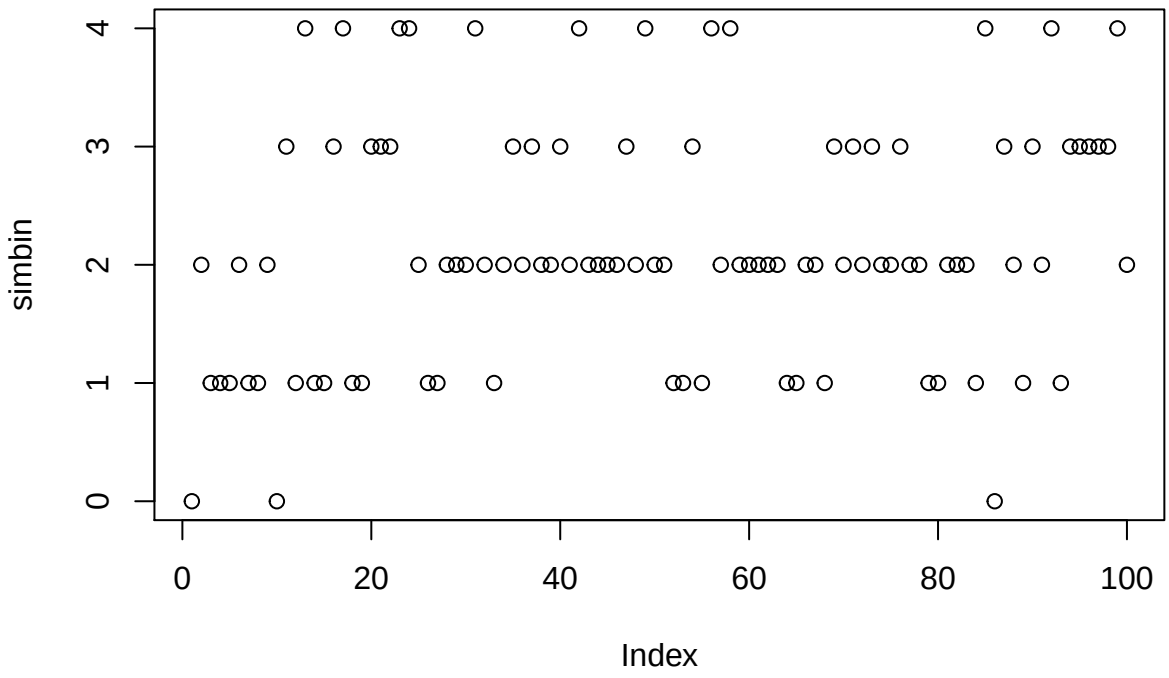
```
pbinom(0:1, 1, .3)
```

```
## [1] 0.7 1.0
```

```
qbinom(probs, 1, .3)
```

```
## [1] 0 0 0 1 1
```

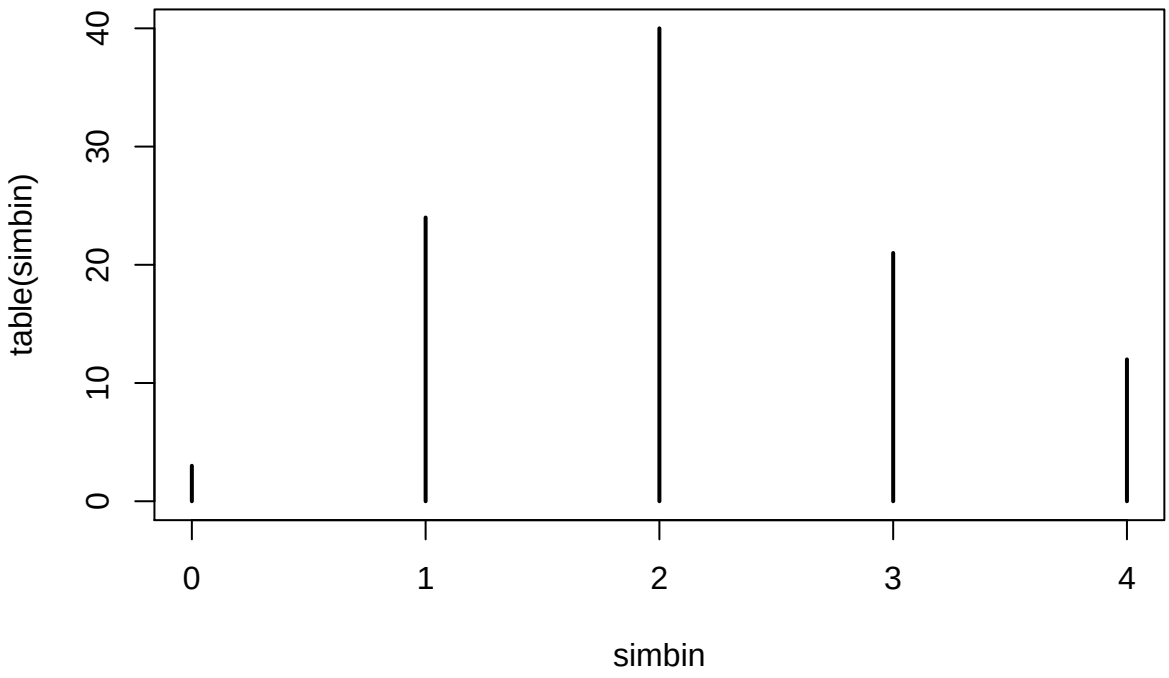
```
simbin <- rbinom(nsim, 4, .5)
plot(simbin)
```




```
table(simbin) / length(simbin)
```

```
## simbin
##    0    1    2    3    4
## 0.03 0.24 0.40 0.21 0.12
```

```
plot(table(simbin))
```




```
dbinom(0:4, 4, .5)
```

```
## [1] 0.0625 0.2500 0.3750 0.2500 0.0625
```

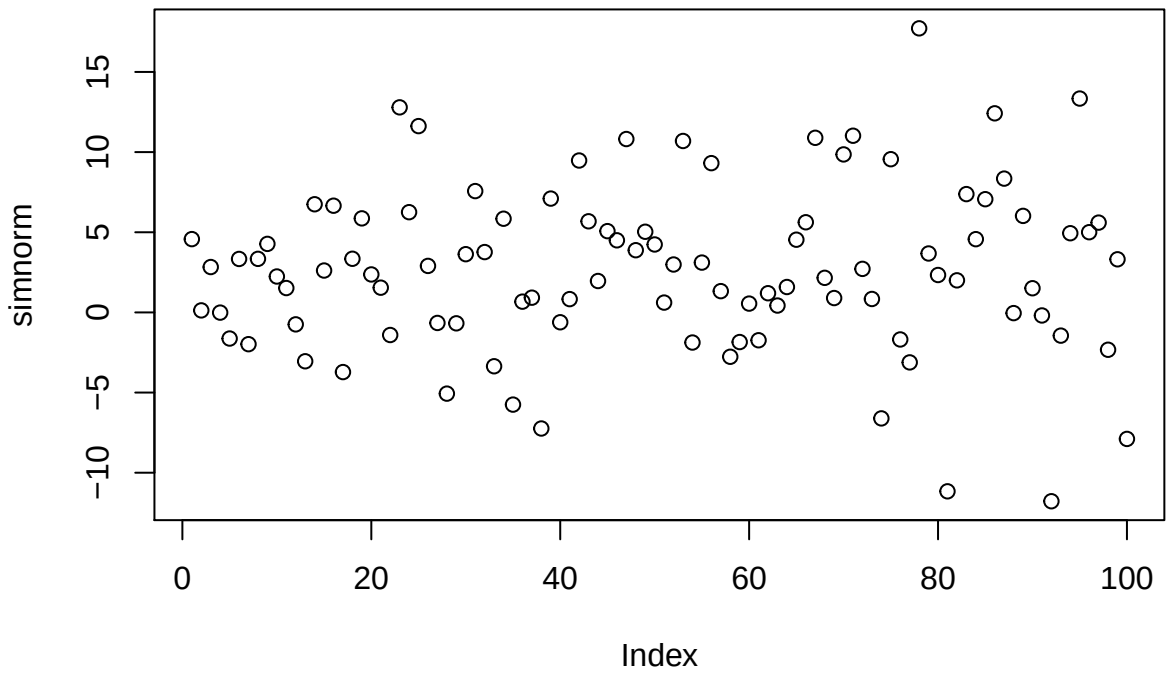
```
pbinom(0:4, 4, .5)
```

```
## [1] 0.0625 0.3125 0.6875 0.9375 1.0000
```

```
qbinom(seq(from = 0, to = 1, length.out = 10), 4, .5)
```

```
##  [1] 0 1 1 2 2 2 2 3 3 4
```

```
simnorm <- rnorm(nsim, mean = 2.5, sd = 5)
plot(simnorm)
```



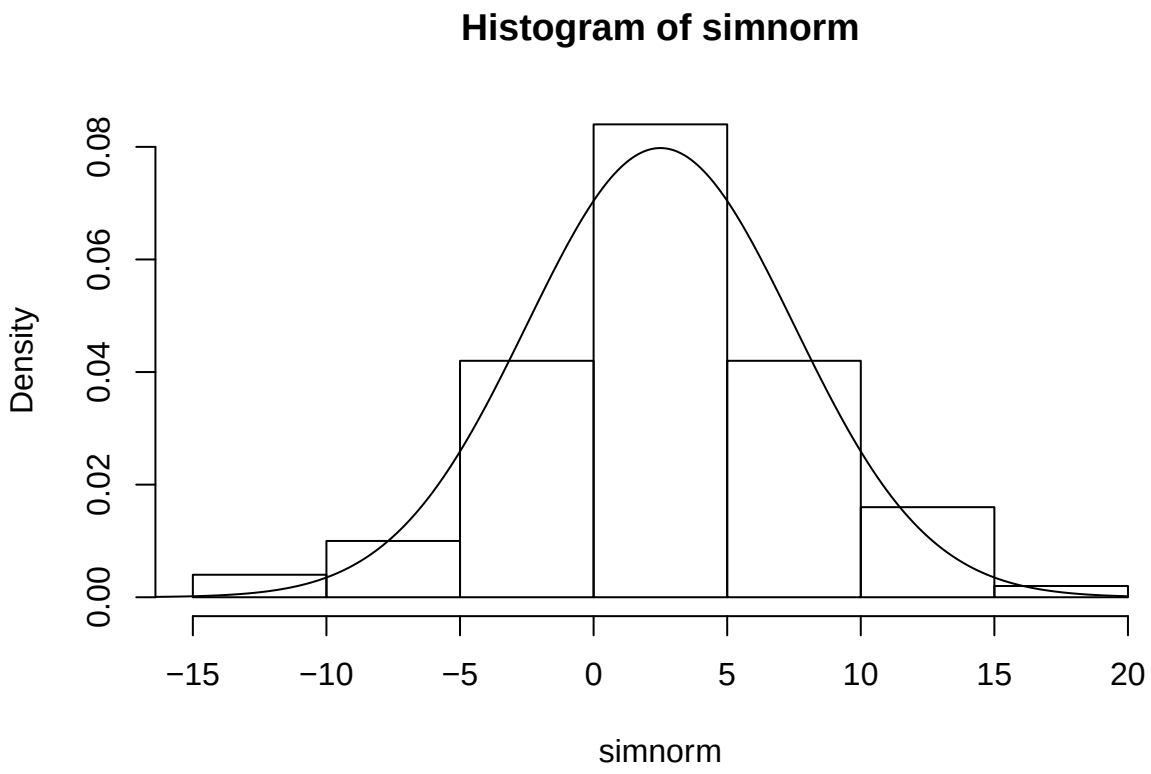

```
dnorm(2.5, mean = 2.5, sd = 5)
```

```
## [1] 0.07978846
```

```
pnorm(2.5, mean = 2.5, sd = 5)
```

```
## [1] 0.5
```

```
ss <- seq(from = - 20, to = 20, by = .01)
hist(simnorm, probability = TRUE)
lines(ss, dnorm(ss, 2.5, 5))
```



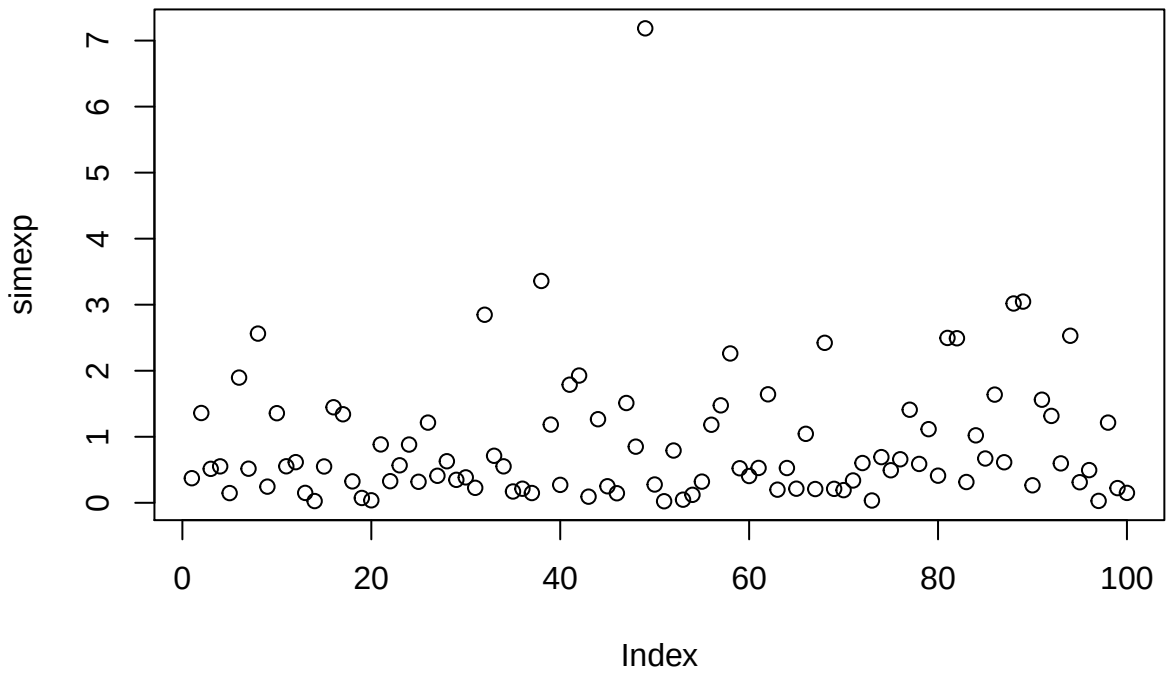

```
summary(simnorm)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -11.7788  -0.2964   2.6640   2.7062   5.6382  17.7188
```

```
qnorm(probs, mean = 2.5, sd = 5)
```

```
## [1]       -Inf -0.8724488  2.5000000  5.8724488        Inf
```

```
simexp <- rexp(nsim, rate = 1)
plot(simexp)
```



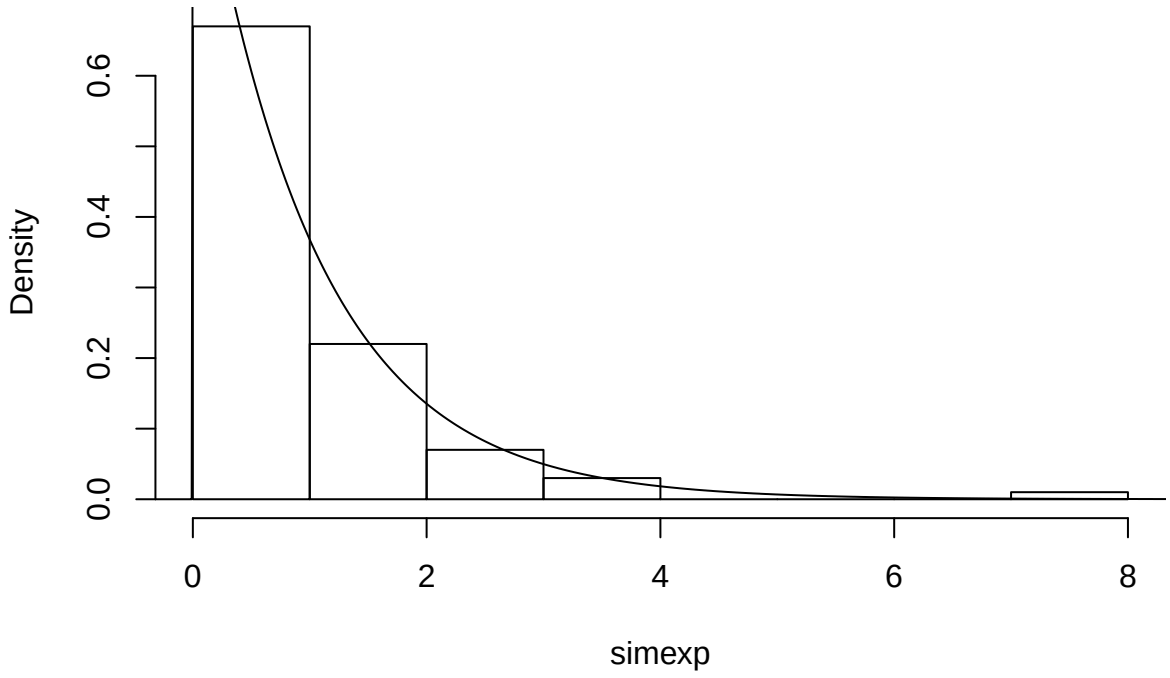

```
dexp(2.5, rate = 1)
```

```
## [1] 0.082085
```

```
pexp(2.5, rate = 1)
```

```
## [1] 0.917915
```

```
hist(simexp, probability = TRUE)
lines(ss, dexp(ss, rate = 1))
```




```
summary(simexp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.02355 0.26141 0.55176 0.90265 1.27820 7.18517
```

```
qexp(probs, rate = 1)
```

```
## [1] 0.0000000 0.2876821 0.6931472 1.3862944       Inf
```