Prácticas de Aprendizaje Automático

Grado en Informática
Doble Grado Informática y Matemáticas

Information

- **Time table :**

  G1 W**(**17,30 – 19,30) class room 3.3

  G2 F**(**17,30 – 19,30) class room 3.3

  G3 W**(**17,30 – 19,30) class room 2.1

- Web page: decsai.ugr.es
- This presentation has been inspired from a tutorial made by

  Roger D. Peng, Associate Professor of Biostatistics
  Johns Hopkins Bloomberg School of Public Health

and the book: Hands-on Programming with R, written by Garrett Grolemund

Plotting Basics

Plotting Basics

The most basic command you can use to produce a plot is **`plot()`**.

The first two arguments to **`plot()`** are taken to be x and y coordinates.

If you only provide one vector of values, this is used as the y-coordinate, and it is plotted against the index values of the vector.

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

Plot Types

There are **9** basic ways for R to plot a set of points. To use one of these, define the **`type`** argument one of these ways:

- p: points (this is the default)
- `l`: a line
- b: "both", points connected by line segments
- c: just the connecting segments of "b"
- o: "overplotted", points and lines overplotted
- h: a histogram
- s: stair
- S: alternative stairs
- n: none

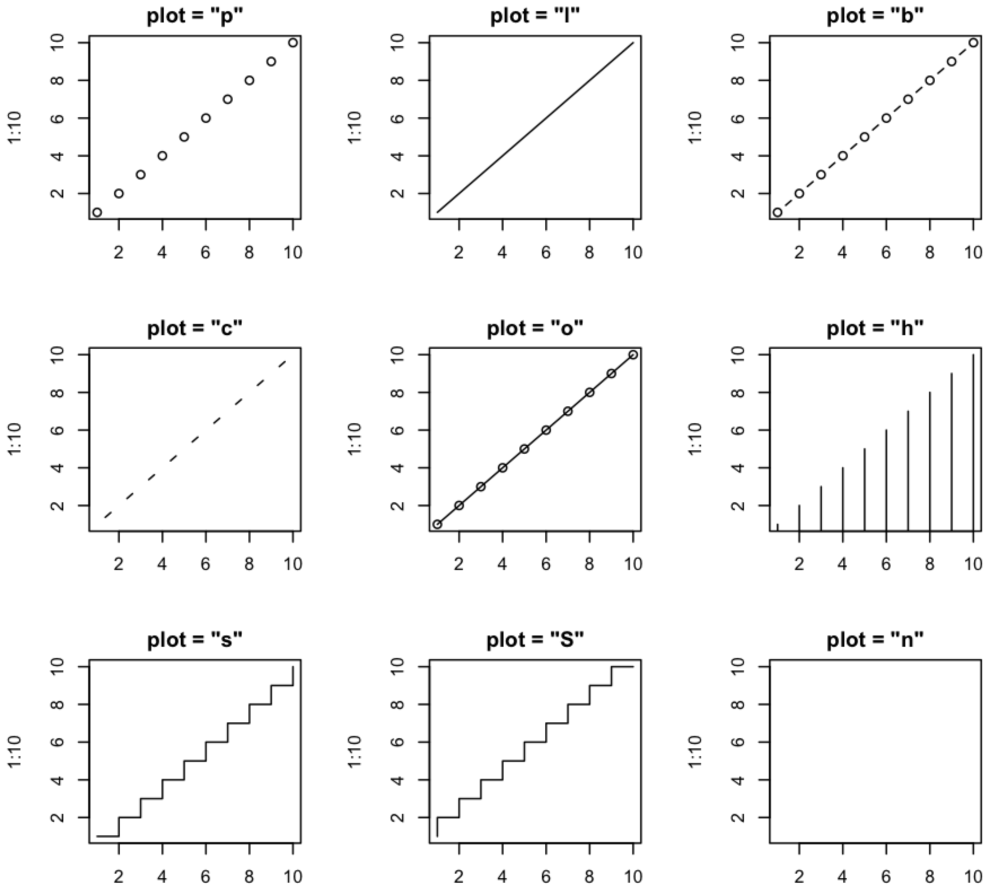
Plot Types

There are 9 basic ways for R to plot a set of points. To use one of these, define the `type` argument one of these ways:

- p: points (this is the default)
- `l`: a line
- b: "both", points connected by line segments
- c: just the connecting segments of "b"
- o: "overplotted", points and lines overplotted
- h: a histogram
- s: stair
- S: alternative stairs
- n: none

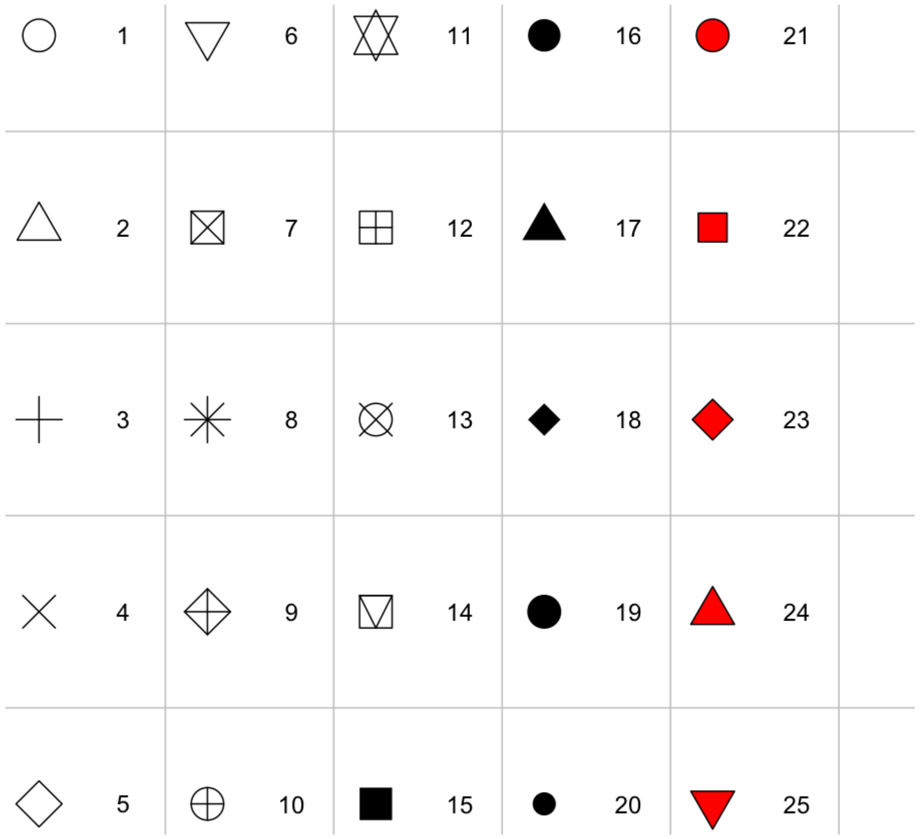
Plotting Basics

```
plot(x, y, type = “p”)
plot(x, y, type = “l”)
plot(x, y, type = “b”)
plot(x, y, type = “c”)
plot(x, y, type = “o”)
plot(x, y, type = “h”)
plot(x, y, type = “s”)
plot(x, y, type = “S”)
plot(x, y, type = “n”)
```



Point Types

There are **25** different plotting points that can be used by defining the argument **pch**

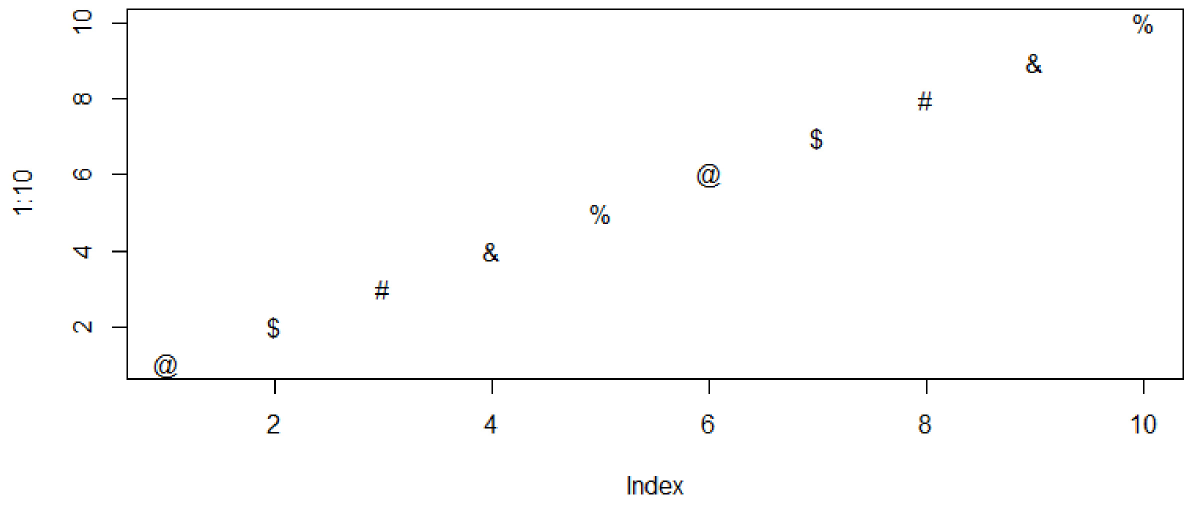


Point Types

There are **25** different plotting points that can be used by defining the argument **pch**

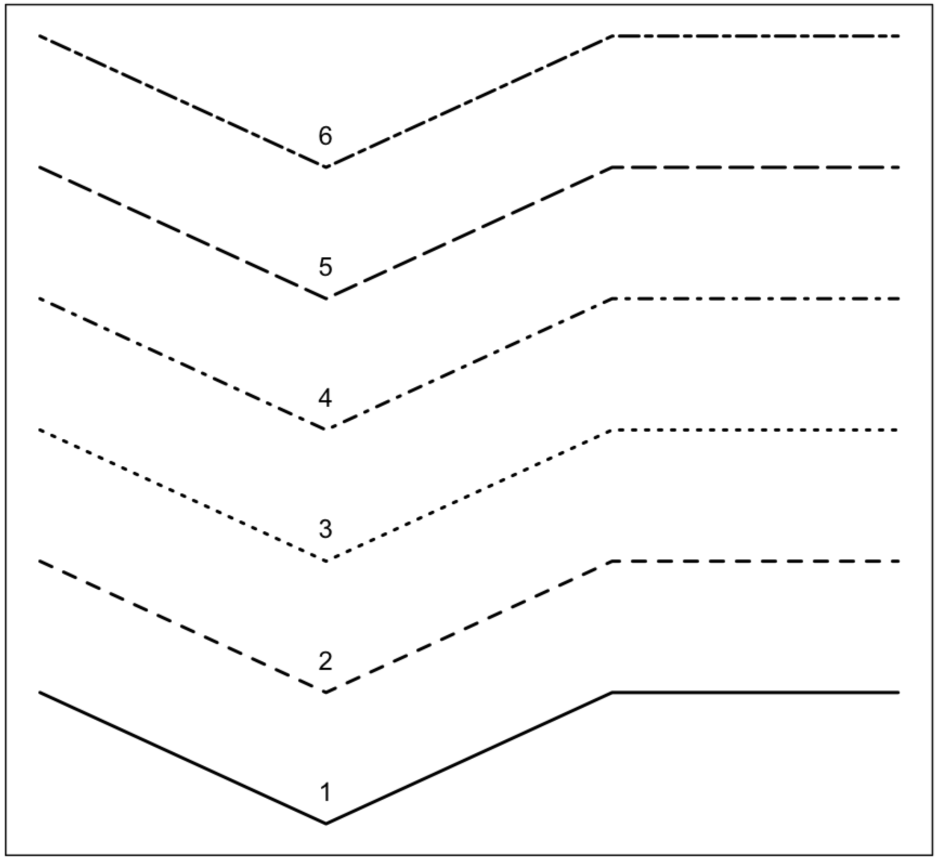
You can also give pch any single character string, and it will plot that.

```
plot(1:10,pch = c("@","$","#","&","%"))
```



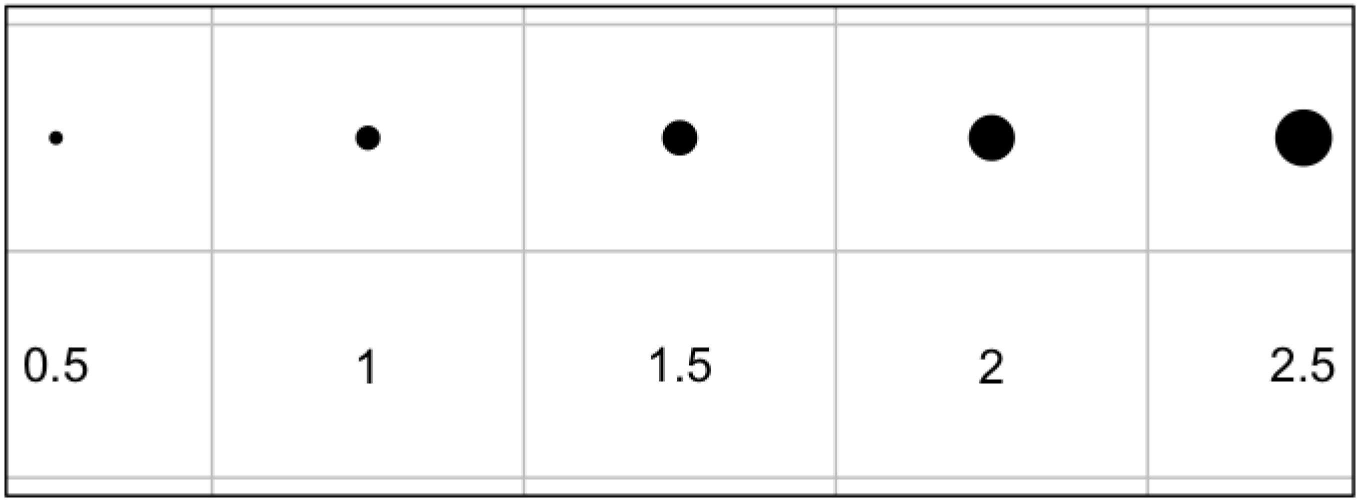
Line Types

There are **6** line types you can use by defining **`lty`** with 1 thru 6.



Weights and Sizes

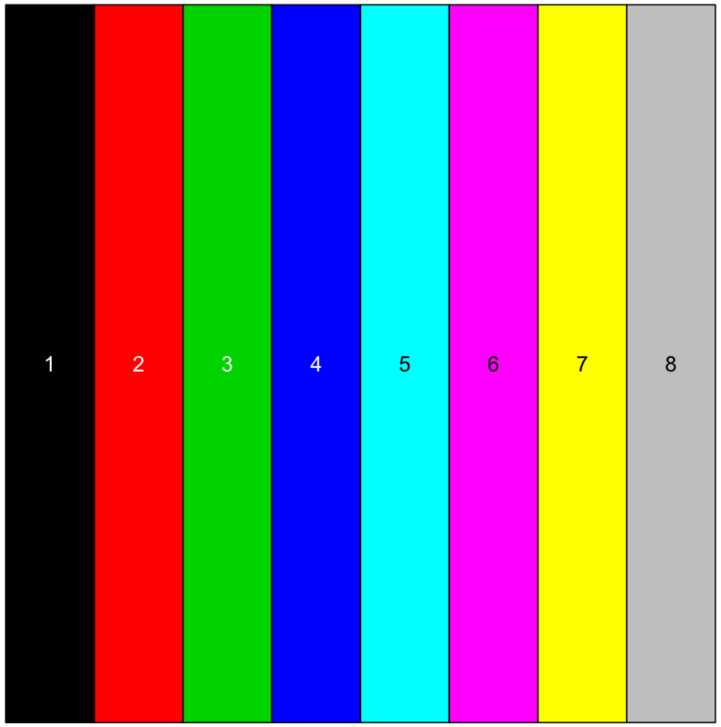
To control the size of plotted points, give the argument **`cex`** a value *x*, which will multiply the size of the character *x* times



There are also `cex` arguments for different plot elements: **`cex.axis`**, **`cex.lab`**, **`cex.main`** and **`cex.sub`**

Colors

R has **8** default system colors, which can be assigned to a point or line by passing a numeric value of 1:8 to **`col`**



Colors

R has **657** named colors you can call upon by passing the **name** of the color to **`col`**. To see all the names, type `colors()`

```
> colors()[1:40]

 [1] "white"          "aliceblue"      "antiquewhite"   "antiquewhite1"
 [5] "antiquewhite2"  "antiquewhite3"  "antiquewhite4"  "aquamarine"
 [9] "aquamarine1"    "aquamarine2"    "aquamarine3"    "aquamarine4"
[13] "azure"          "azure1"         "azure2"         "azure3"
[17] "azure4"         "beige"          "bisque"         "bisque1"
[21] "bisque2"        "bisque3"        "bisque4"        "black"
[25] "blanchedalmond" "blue"           "blue1"          "blue2"
[29] "blue3"          "blue4"          "blueviolet"     "brown"
[33] "brown1"         "brown2"         "brown3"         "brown4"
[37] "burlywood"      "burlywood1"     "burlywood2"     "burlywood3"
```

Colors

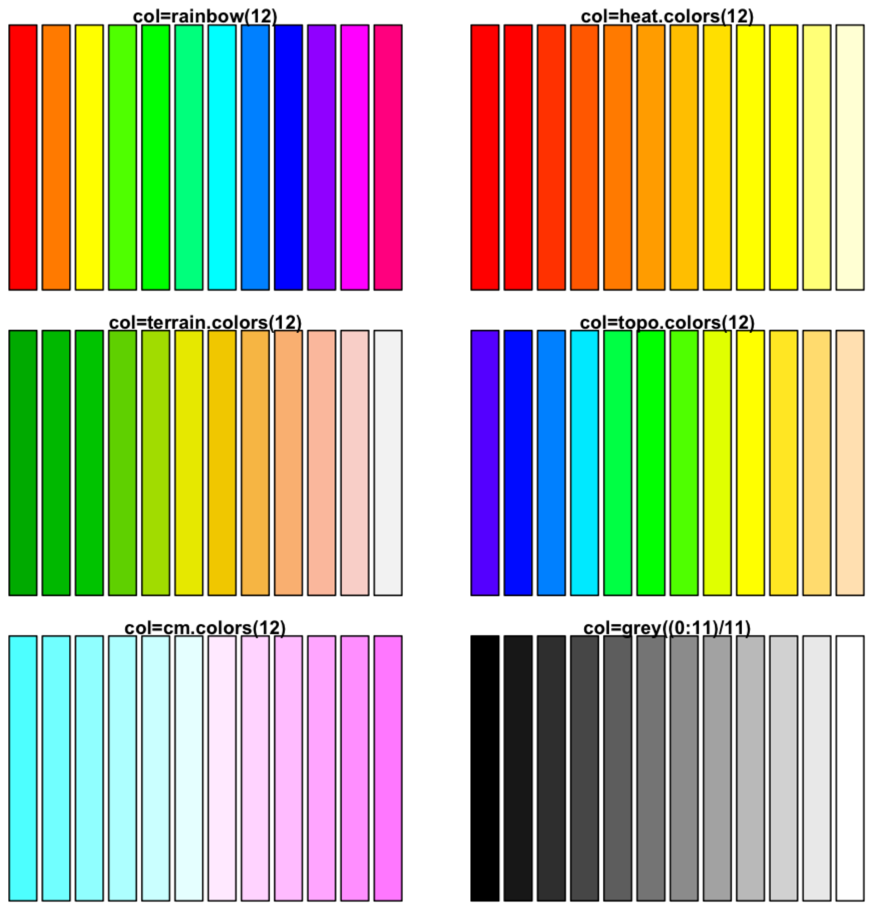
There are particular color palettes that you could select colors from as well:

- `rainbow()`
- `heat.colors()`
- `terrain.colors()`
- `topo.colors()`
- `cm.colors()`
- `Grey()`

```
plot(x,y, type= "o",pch=3, col= rainbow(3))
```

Colors

There are particular color palettes that you could select colors from as well:



Setting background

Setting background

The simplest way to think about it is that first you define the paper on which the plots will be drawn, then you define the plotting region in which you will draw, and then you add elements to that region.

Par()

`par()` stands for "graphical **par**ameters", and in essence defines the paper on which plots will be drawn, as well as certain parameters which will be inherited by each plot on the page.

Most graphical parameters can be set for each individual plot, but some must be set for the whole page. To see the full list of definable parameters, see **`?par`**

mfcol and mfrow

To have multiple plots per page, we define it as follows:

```
n.col <- 2
n.row <- 2
par(mfrow = c(n.row,n.col))
```

`mfrow`

Plot1	**Plot2**
Plot3	**Plot4**

`mfcol`

Plot1	**Plot3**
Plot2	**Plot4**

mar

The **mar** argument sets the margins in terms of "lines" between the plotting region and the edge of the page. By default, they are set to `c(5,4,4,2)+0.1`

The numbers in the vectors stand for c(bottom, left, top, right).

bg

When defined within **`par()`**`,` **`bg`** changes the background color of the plotting area. By default, this is set to `transparent`.

Plot properties

Plot properties

Assuming you have set up your page as you like, then you need to define the plotting region with **`plot()`**

x and y limits

By default, R will set the limits of the `x` and `y` axes to accommodate the range of `x` and `y` values passed to it. However, you can hand adjust the range of the axes with **`xlim`** and **`ylim`**. They take a vector of two numerical values: `c(from,to).`

Useful functions:

- `range()`will return the range of values
- `rev()` will reverse them

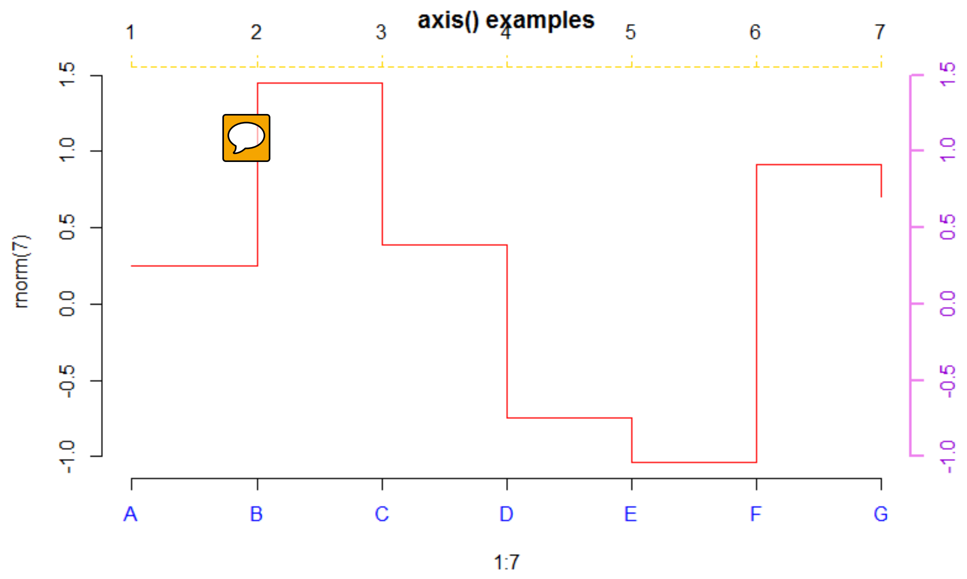
Axes

By default **`plot()`** will draw axes with ticks and labels at automatically determined places. If you would like to hand roll your own axes, add properties to the axes with `axis()`.

```
plot(1:7, rnorm(7), main = "axis() examples", type = "s",
xaxt = "n", frame = FALSE, col = "red")

axis(1, 1:7, LETTERS[1:7], col.axis = "blue")
# unusual options:

axis(4, col = "violet", col.axis="dark violet", lwd = 2)
axis(3, col = "gold", lty = 2, lwd = 0.5)
```

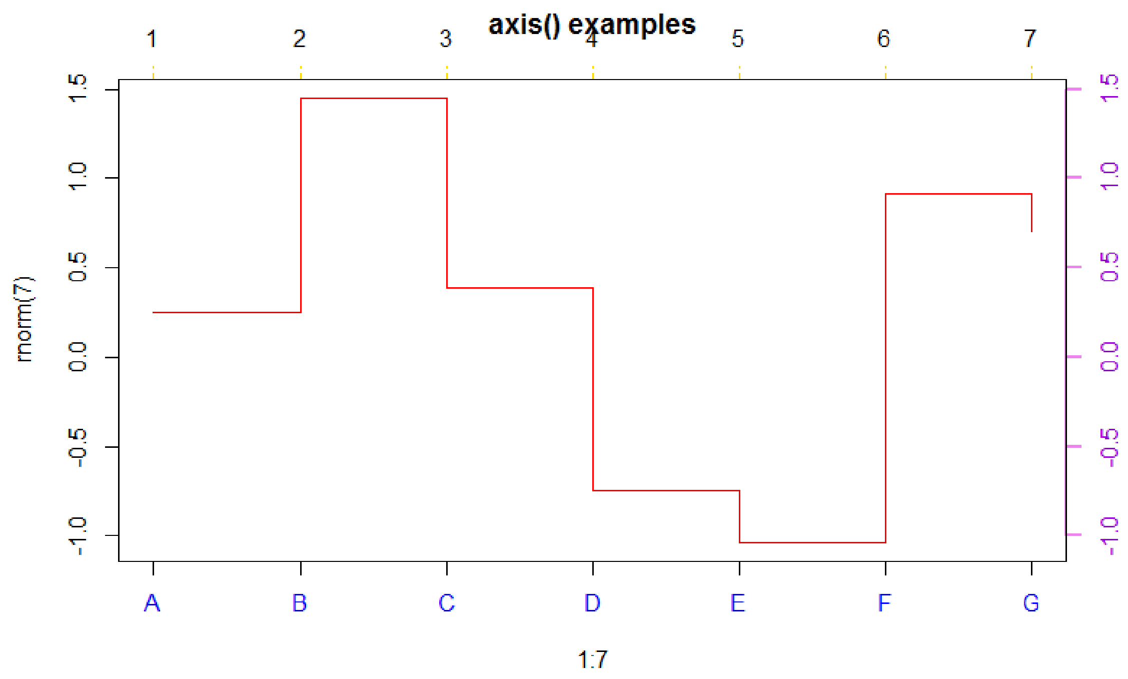
Axes

```
plot(1:7, rnorm(7), main = "axis() examples", type = "s",
xaxt = "n", frame = FALSE, col = "red")

axis(1, 1:7, LETTERS[1:7], col.axis = "blue")
# unusual options:

axis(4, col = "violet", col.axis="dark violet", lwd = 2)
axis(3, col = "gold", lty = 2, lwd = 0.5)
```



Bounding box

Using function `box()` we can plot a bounding box around the axis.



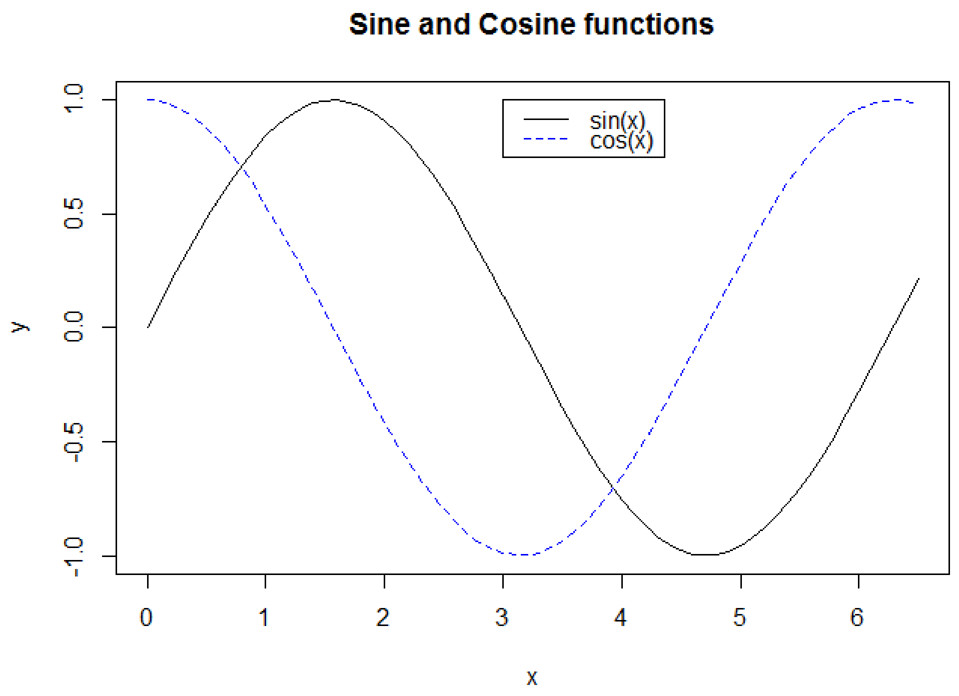
Titles and Axis labels

There are some parameters of function **`plot()`**, to write the titles, subtitles and axis names:

- `main :` It writes the figure TITLE
- `sub  :` It writes the figure SUBTITLE
- `xlab :` It writes x-axis label
- `ylab :` It writes x-axis label

Legends

```
x <- 0:65/10
y <- sin(x)
z <- cos(x)
plot(x,y,main= "Sine and Cosine functions", type = "l")
lines(x,z,col="blue",lty=2)
legend(x=3,y=1,legend=c("sin(x)"," cos(x)"),lty=c(1,2),col=c("black","blue"))
```



Higher Level Plots

Other plots

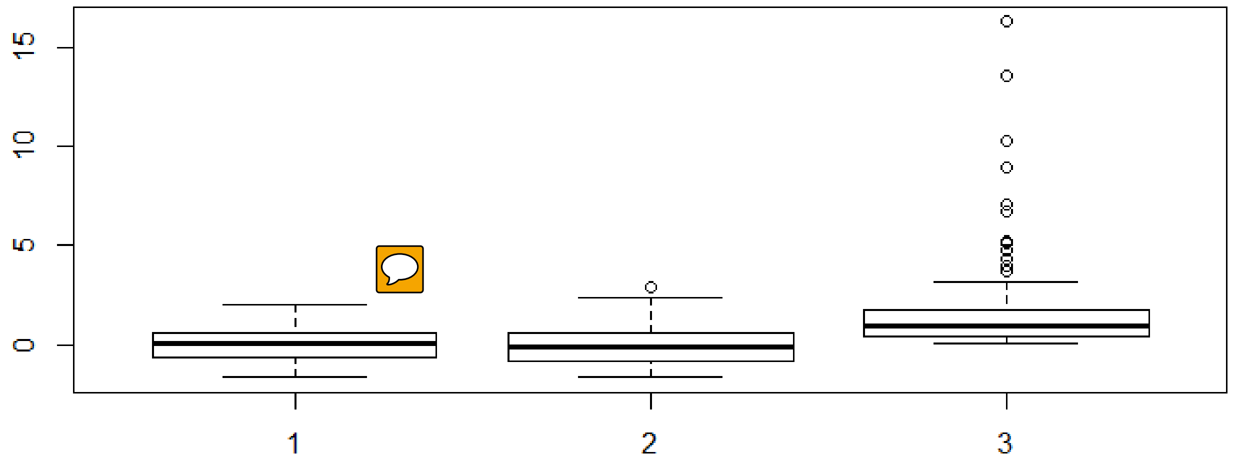
In addition of **`plot(),`** R has more functions to higher level plots:

- Box Plots
- Histogram
- Pie ...

Box plots

The **box** indicates the location of the first and third quartile, with the median indicated with a bar. Whiskers are drawn out to 1.5 times the length of the box, and any points outside this range are plotted (these can be considered outliers).

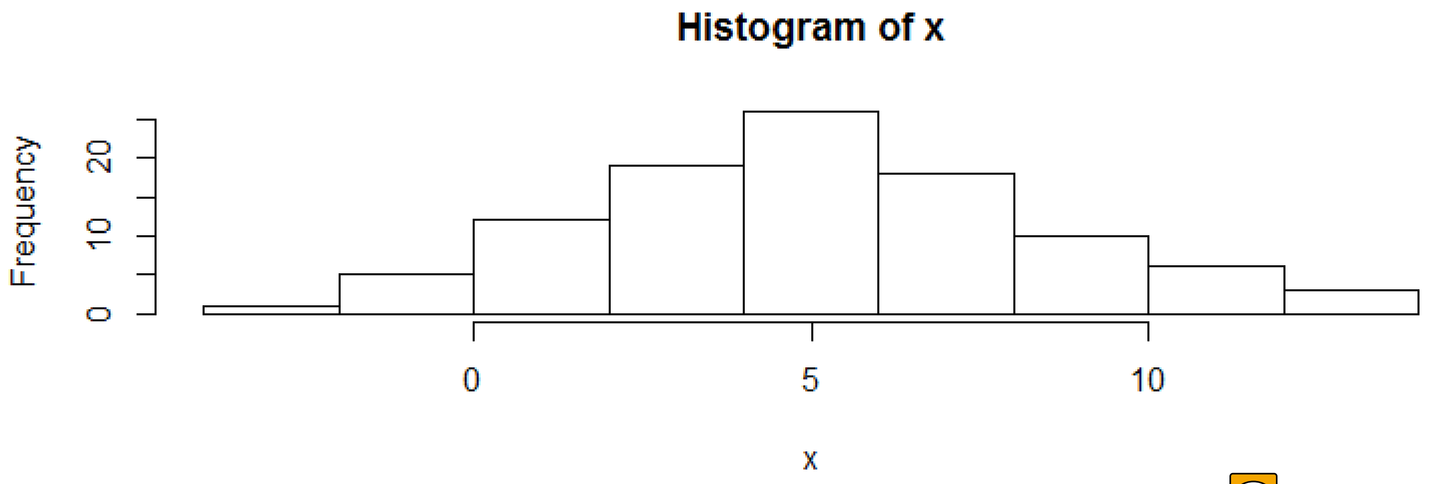
```
boxplot(rnorm(50),rnorm(100),rlnorm(100))
```



Histogram

The histogram of a dataset can be ploted with **`hist()`** function

```
x <- rnorm(100,5,3)
hist(x)
```



Loops and Functions

Vectorized Code

We can write a piece of code in many different ways, but the fastest R code will usually take advantage of three things: logical tests, subsetting, and element-wise execution. Let's compare two different ways to solve the same problem:

```
abs_loop <- function(vec){
    for (i in 1:length(vec)) {
          if (vec[i] < 0) {
               vec[i] <- -vec[i]
          }
    }
vec
}
```

abs_loop uses a **for** loop to manipulate each element of the vector one at a time.

Vectorized Code

The second version is what it known as vectorized code

```
abs_sets <- function(vec){
    negs <- vec < 0
    vec[negs] <- vec[negs] * -1
    vec
}
```

`abs_set` is much faster than **`abs_loop,`** how much?
To measure it, we will use a long vector as :

```
> long <- rep(c(-1, 1), 5000000)
```

Vectorized Code or not vectorized?

```
> System.time(abs_loop(long))
## user system elapsed
## 15.982 0.032 16.018
> system.time(abs_sets(long))
## user system elapsed
## 0.529 0.063 0.592
```

The results show that **abs_set** calculated the absolute value 30 times faster than **abs_loop** we can speed-up even more by using the appropriate funcion …

```
> system.time(abs(long))
## user system elapsed
## 0.037 0.018 0.054
```

How to Write Vectorized Code

Vectorized code is easy to write in R because most R functions are already vectorized. Code based on these functions can easily be made vectorized and therefore fast.

To create vectorized code:

1. Use vectorized functions to complete the sequential steps in your program.
2. Use logical subsetting to handle parallel cases. Try to manipulate every element in a case at once.

```
If a number is positive,
  the functions leave it alone.
If a number is negative, the functions multiply it by
negative one.
```

How to Write Vectorized Code

vec < 0 identifies every value of **vec** that belongs to the negative case. You can use the same logical test to extract the set of negative values with logical subsetting

```
vec <- c(1, -2, 3, -4, 5, -6, 7, -8, 9, -10)
vec < 0
## FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
vec[vec < 0]
## -2 -4 -6 -8 -10
vec[vec < 0] * -1
## 2 4 6 8 10
vec[vec < 0] <- vec[vec < 0] * -1
```

All of R's arithmetic operators are vectorized, so you can use * to complete the step in vectorized fashion. Finally, use R's assignment operator, which is also vectorized, to save the new set over the old set in the original vec object. Since <- is vectorized, the elements of the new set will be paired up to the elements of the old set

Vectorized Code or not vectorized?

```
change_forif <- function(vec){
   for (i in 1:length(vec)){
      if (vec[i] == "DD") {
         vec[i] <- "joker"
      } else if (vec[i] == "C") {
         vec[i] <- "ace"
      } else if (vec[i] == "7") {
         vec[i] <- "king"
      }else if (vec[i] == "B") {
         vec[i] <- "queen"
      } else if (vec[i] == "BB") {
         vec[i] <- "jack"
      } else if (vec[i] == "BBB") {
         vec[i] <- "ten"
      } else {
         vec[i] <- "nine"
      }
   }
Vec } #end function
```

Vectorized Code or not vectorized?

```
change_vec <- function (vec) {
   vec[vec == "DD"] <- "joker"
   vec[vec == "C"] <- "ace"
   vec[vec == "7"] <- "king"
   vec[vec == "B"] <- "queen"
   vec[vec == "BB"] <- "jack"
   vec[vec == "BBB"] <- "ten"
   vec[vec == "0"] <- "nine"
vec
}

vec <- c("DD", "C", "7", "B", "BB", "BBB", "0")

change_forif(vec)  # or change_vec(vec)
## "joker" "ace" "king" "queen" "jack" "ten" "nine"
```

Vectorized Code or not vectorized?

```
many <- rep(vec, 1000000) # many symbols to convert

> system.time(change_forif(many))
## user system elapsed
## 30.057 0.031 30.079

> system.time(change_vec(many))
## user system elapsed
## 1.994 0.059 2.051
```

How to Write Vectorized Code

To vectorize ***change_forif*** , create a logical test that can identify each case: like **vec[vec == "DD"]**
Then write code that can change the symbols for each case:
vec[vec == "DD"] <- "joker" .
When you combine this into a function, you have a vectorized version of ***change_forif*** that runs about 14 times faster.

In R, without the compilation step that optimizes how the for loops in the code use the computer’s memory, which makes the for loops very fast loops; the loops will run slower than the other operations we have studied: logical tests, subsetting, and element-wise execution.

for loops do not behave the same way in R as they do in other languages, which means to write code differently in R than in other languages.

How to Write Vectorized Code

if and **for**

A good way to spot for loops that could be vectorized is to look for combinations of **if** and **for** .
if can only be applied to one value at a time, which means it is often used in conjunction with a **for** loop.
The **for** loop helps apply **if** to an entire vector of values.

This combination can usually be replaced with **logical subsetting**, which will do the same thing but run **much faster**.

Loop functions

Looping in the command line

Writing **for**, **while** loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier. An auxiliary function split is also useful, particularly in conjunction with lapply.

- lapply: Loop over a list and evaluate a function on each element

- sapply: Same as lapply but try to simplify the result

- apply: Apply a function over the margins of an array

- tapply: Apply a function over subsets of a vector

- mapply: Multivariate version of lapply

lapply

lapply always returns a list, regardless of the class of the input.

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.4671
```

lapply

lapply always returns a list, regardless of the class of the input.

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d =
rnorm(100, 5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] 0.5261
##
## $c
## [1] 1.421
##
## $d
## [1] 4.927
```

lapply

lapply always returns a list, regardless of the class of the input.

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.2675082

[[2]]
[1] 0.2186453 0.5167968

[[3]]
[1] 0.2689506 0.1811683 0.5185761

[[4]]
[1] 0.5627829 0.1291569 0.2563676 0.7179353
```

lapply

lapply always returns a list, regardless of the class of the input.

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 3.302142

[[2]]
[1] 6.848960 7.195282

[[3]]
[1] 3.5031416 0.8465707 9.7421014

[[4]]
[1] 1.195114 3.594027 2.930794 2.766946
```

lapply

lapply and friends make heavy use of anonymous functions.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6,
3, 2))
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

lapply

An anonymous function for extracting the first column of each matrix.

```
> lapply(x, function(elt) elt[,1])
$a
[1] 1 2

$b
[1] 1 2 3
```

sapply

sapply will try to simplify the result of lapply if possible.

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

sapply

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d =
rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.06082667

$c
[1] 1.467083

$d
[1] 5.074749
```

sapply

```
> sapply(x, mean)
         a          b          c          d 
2.50000000 0.06082667 1.46708277 5.07474950 

> mean(x)
[1] NA
Warning message:
In mean.default(x) : argument is not numeric or logical: 
returning NA
```

apply

apply is used to a evaluate a function (often an anonymous one) over the margins of an array.

It is most often used to apply a function to the rows or columns of a matrix

It can be used with general arrays, e.g. taking the average of an array of matrices

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- X is an array

- MARGIN is an integer vector indicating which margins should be “retained”.

- FUN is a function to be applied

- ... is for other arguments to be passed to FUN

apply

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
 [1]  0.04868268  0.35743615 -0.09104379
 [4] -0.05381370 -0.16552070 -0.18192493
 [7]  0.10285727  0.36519270  0.14898850
[10]  0.26767260

> apply(x, 1, sum)
 [1] -1.94843314  2.60601195  1.51772391
 [4] -2.80386816  3.73728682 -1.69371360
 [7]  0.02359932  3.91874808 -2.39902859
[10]  0.48685925 -1.77576824 -3.34016277
[13]  4.04101009  0.46515429  1.83687755
[16]  4.36744690  2.21993789  2.60983764
[19] -1.48607630  3.58709251
```

apply

For sums and means of matrix dimensions, we have some shortcuts.

rowSums = apply(x, 1, sum)

rowMeans = apply(x, 1, mean)

colSums = apply(x, 2, sum)

colMeans = apply(x, 2, mean)

The shortcut functions are much faster, but you won’t notice unless you’re using a large matrix.