MAP 2112 – Introdução à Lógica de Programação e Modelagem Computacional

1º Semestre - 2018

Prof. Dr. Luis Carlos de Castro Santos

lsantos@ime.usp.br/lccs13@yahoo.com

Functions

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

Functions

Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```
f <- function(<arguments>) {
        ## Do something interesting
}
```

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated.

Function Arguments

Functions have *named arguments* which potentially have *default values*.

- The *formal arguments* are the arguments included in the function definition
- The `formals` function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be *missing* or might have default values

sd = standard deviation = desvio padrão

Argument Matching

R functions arguments can be matched positionally or by name. So the following calls to **sd** are all equivalent

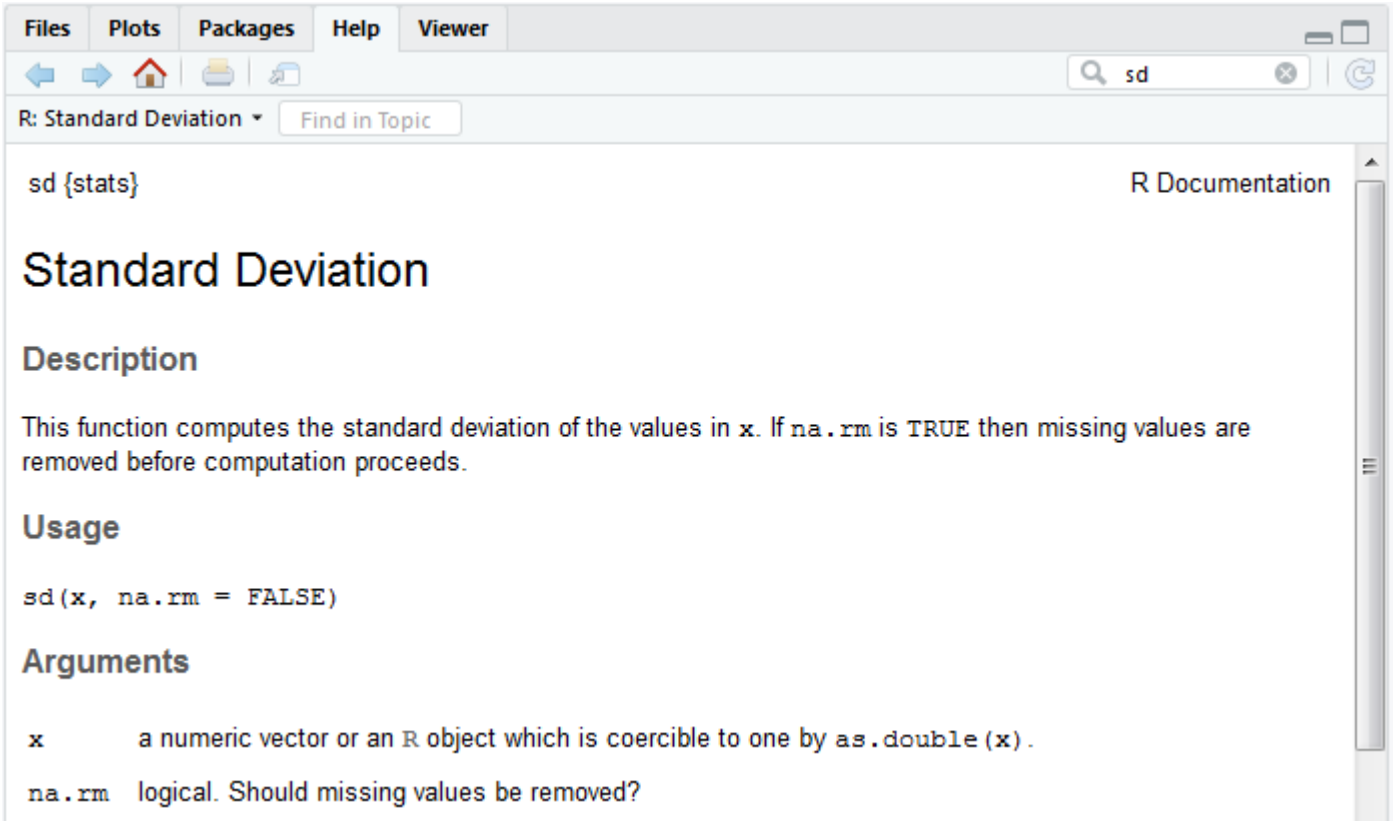
```
> mydata <- rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

```
> formals(sd)
$x


$na.rm
[1] FALSE
```

```
> args(sd)
function (x, na.rm = FALSE)
NULL
```



Argument Matching

You can mix positional matching with matching by name. When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> args(lm)
function (formula, data, subset, weights, na.action,
          method = "qr", model = TRUE, x = FALSE,
          y = FALSE, qr = TRUE, singular.ok = TRUE,
          contrasts = NULL, offset, ...)
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

Argument Matching

- Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
- Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Function arguments can also be *partially* matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Defining a Function

```
f <- function(a, b = 1, c = 2, d = NULL) {

}
```

In addition to not specifying a default value, you can also set an argument value to `NULL`.

A First Example

exemplo1.R | Untitled1* | function_scripts.R*

Source on Save

```
1
2
3
4  add2 <- function(x,y){
5          x + y
6  }
7
8  above10 <- function(x){
9            use <- x > 10
10           x[use]
11 }
12
13 above <- function(x, n){
14           use <- x > n
15           x[use]
16 }
17
18 above <- function(x, n = 10){
19   use <- x > n
20   x[use]
21 }
```

default

```
> add2(2,3)
[1] 5
```

```
> x <- 1:20
> above10(x)
[1] 11 12 13 14 15 16 17 18 19 20
```

```
> above(x,6)
[1] 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
> above(x)
[1] 11 12 13 14 15 16 17 18 19 20
```

A Second (more interesting) Example

functions_aula06.R*

Source on Save

```
1  columnmean <- function(y){
2                nc <- ncol(y)
3                means <- numeric(0)
4                for (i in 1:nc) {
5                  means[i] <- mean(y[,i])
6                }
7                means
8  }
9
```

Apenas para definir o tipo do objeto

```
> columnmean(airquality)
[1]        NA        NA  9.957516 77.882353  6.993464 15.803922
```

A Second (more interesting) Example (ii)

```
columnmean2 <- function(y, removeNA = TRUE){
  nc <- ncol(y)
  means <- numeric(0)
  for (i in 1:nc) {
    means[i] <- mean(y[,i], na.rm = removeNA)
  }
  means
}
```

```
> columnmean2(airquality)
[1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed.

```
f <- function(a, b) {
    a^2
}
f(2)
```

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

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the 2 gets positionally matched to `a`.

Lazy Evaluation

```
f <- function(a, b) {
    print(a)
    print(b)
}
f(45)
```

```
## [1] 45
```

```
## Error: argument "b" is missing, with no default
```

Notice that "45" got printed first before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` it had to throw an error.

The "..." Argument

The ... argument indicate a variable number of arguments that are usually passed on to other functions.

- ... is often used when extending another function and you don't want to copy the entire argument list of the original function

```
myplot <- function(x, y, type = "l", ...) {
        plot(x, y, type = type, ...)
}
```

A Second (more interesting) Example (iii)

```
columnmean3 <- function(y, ...){
  nc <- ncol(y)
  means <- numeric(0)
  for (i in 1:nc) {
    means[i] <- mean(y[,i], ...)
  }
  means
}
```

```
> columnmean3(airquality, na.rm = TRUE)
[1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

The "..." Argument

The ... argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> args(cat)
function (..., file = "", sep = " ", fill = FALSE,
    labels = NULL, append = FALSE)
```

Arguments Coming After the "..." Argument

One catch with ... is that any arguments that appear *after* ... on the argument list must be named explicitly and cannot be partially matched.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> paste("a", "b", sep = ":")
[1] "a:b"

> paste("a", "b", se = ":")
[1] "a b :"
```

Aplicações

Condicionais
Laços
Funções

Problema 1

Escreva um código em R para encontrar as raízes de uma equação do 2º grau dados os coeficientes da equação.

$$ax^2 + bx + c = 0$$

Base R
Cheat Sheet

Getting Help

Accessing the help files

?mean
Get help of a particular function.
help.search('weighted mean')
Search the help files for a word or phrase.
help(package = 'dplyr')
Find help for a package.

More about an object

str(iris)
Get a summary of an object's structure.
class(iris)
Find the class an object belongs to.

Using Packages

install.packages('dplyr')
Download and install a package from CRAN.

library(dplyr)
Load the package into the session, making all its functions available to use.

dplyr::select
Use a particular function from a package.

data(iris)
Load a built-in dataset into the environment.

Working Directory

getwd()
Find the current working directory (where inputs are found and outputs are sent).

setwd('C://file/path')
Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

Vectors

Creating Vectors

`c(2, 4, 6)`	2 4 6	Join elements into a vector
`2:6`	2 3 4 5 6	An integer sequence
`seq(2, 3, by=0.5)`	2.0 2.5 3.0	A complex sequence
`rep(1:2, times=3)`	1 2 1 2 1 2	Repeat a vector
`rep(1:2, each=3)`	1 1 1 2 2 2	Repeat elements of a vector

Vector Functions

sort(x)
Return x sorted.
rev(x)
Return x reversed.
table(x)
See counts of values.
unique(x)
See unique values.

Selecting Vector Elements

By Position

`x[4]`	The fourth element.
`x[-4]`	All but the fourth.
`x[2:4]`	Elements two to four.
`x[-(2:4)]`	All elements except two to four.
`x[c(1, 5)]`	Elements one and five.

By Value

`x[x == 10]`	Elements which are equal to 10.
`x[x < 0]`	All elements less than zero.
`x[x %in% c(1, 2, 5)]`	Elements in the set 1, 2, 6.

Named Vectors

`x['apple']`	Element with name 'apple'.

Programming

For Loop

```
for (variable in sequence){
   Do something
}
```

Example

```
for (i in 1:4){
   j <- i + 10
   print(j)
}
```

While Loop

```
while (condition){
   Do something
}
```

Example

```
while (i < 5){
   print(i)
   i <- i + 1
}
```

If Statements

```
if (condition){
   Do something
} else {
   Do something different
}
```

Example

```
if (i > 3){
   print('Yes')
} else {
   print('No')
}
```

Functions

```
function_name <- function(var){
   Do something
   return(new_variable)
}
```

Example

```
square <- function(x){
   squared <- x*x
   return(squared)
}
```

Reading and Writing Data

Also see the **readr** package.

Input	Ouput	Description
`df <- read.table('file.txt')`	`write.table(df, 'file.txt')`	Read and write a delimited text file.
`df <- read.csv('file.csv')`	`write.csv(df, 'file.csv')`	Read and write a comma separated value file. This is a special case of read.table/ write.table.
`load('file.RData')`	`save(df, file = 'file.Rdata')`	Read and write an R data file, a file type special for R.

Conditions

`a == b`	Are equal	`a > b`	Greater than	`a >= b`	Greater than or equal to	`is.na(a)`	Is missing
`a != b`	Not equal	`a < b`	Less than	`a <= b`	Less than or equal to	`is.null(a)`	Is null

Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

`as.logical`	`TRUE, FALSE, TRUE`	Boolean values (TRUE or FALSE).
`as.numeric`	`1, 0, 1`	Integers or floating point numbers.
`as.character`	`'1', '0', '1'`	Character strings. Generally preferred to factors.
`as.factor`	`'1', '0', '1', levels: '1', '0'`	Character strings with preset levels. Needed for some statistical models.

Maths Functions

`log(x)`	Natural log.	`sum(x)`	Sum.
`exp(x)`	Exponential.	`mean(x)`	Mean.
`max(x)`	Largest element.	`median(x)`	Median.
`min(x)`	Smallest element.	`quantile(x)`	Percentage quantiles.
`round(x, n)`	Round to n decimal places.	`rank(x)`	Rank of elements.
`signif(x, n)`	Round to n significant figures.	`var(x)`	The variance.
`cor(x, y)`	Correlation.	`sd(x)`	The standard deviation.

Variable Assignment

```
> a <- 'apple'
> a
[1] 'apple'
```

The Environment

`ls()`	List all variables in the environment.
`rm(x)`	Remove x from the environment.
`rm(list = ls())`	Remove all variables from the environment.

You can use the environment panel in RStudio to browse variables in your environment.

Matrices

`m <- matrix(x, nrow = 3, ncol = 3)`
Create a matrix from x.


`m[2, ]` - Select a row


`m[ , 1]` - Select a column


`m[2, 3]` - Select an element

`t(m)`
Transpose

`m %*% n`
Matrix Multiplication

`solve(m, n)`
Find x in: m * x = n

Lists

`l <- list(x = 1:5, y = c('a', 'b'))`
A list is a collection of elements which can be of different types.

`l[[2]]`	`l[1]`	`l$x`	`l['y']`
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

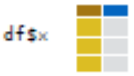
Also see the **dplyr** package.

Data Frames

`df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))`
A special case of a list where all elements are the same length.

x	y
1	a
2	b
3	c

List subsetting

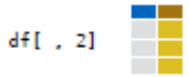

`df$x` `df[[2]]`

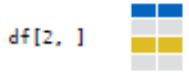
Understanding a data frame

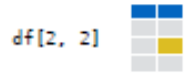
`View(df)` See the full data frame.

`head(df)` See the first 6 rows.

Matrix subsetting


`df[ , 2]`


`df[2, ]`


`df[2, 2]`

`nrow(df)` Number of rows.

`ncol(df)` Number of columns.

`dim(df)` Number of columns and rows.

`cbind` - Bind columns.

`rbind` - Bind rows.

Strings

Also see the **stringr** package.

`paste(x, y, sep = ' ')`	Join multiple vectors together.
`paste(x, collapse = ' ')`	Join elements of a vector together.
`grep(pattern, x)`	Find regular expression matches in x.
`gsub(pattern, replace, x)`	Replace matches in x with a string.
`toupper(x)`	Convert to uppercase.
`tolower(x)`	Convert to lowercase.
`nchar(x)`	Number of characters in a string.

Factors

`factor(x)`
Turn a vector into a factor. Can set the levels of the factor and the order.

`cut(x, breaks = 4)`
Turn a numeric vector into a factor by 'cutting' into sections.

Statistics

`lm(y ~ x, data=df)`
Linear model.

`glm(y ~ x, data=df)`
Generalised linear model.

`summary`
Get more detailed information out a model.

`t.test(x, y)`
Perform a t-test for difference between means.

`pairwise.t.test`
Perform a t-test for paired data.

`prop.test`
Test for a difference between proportions.

`aov`
Analysis of variance.

Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	`rnorm`	`dnorm`	`pnorm`	`qnorm`
Poisson	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** package.

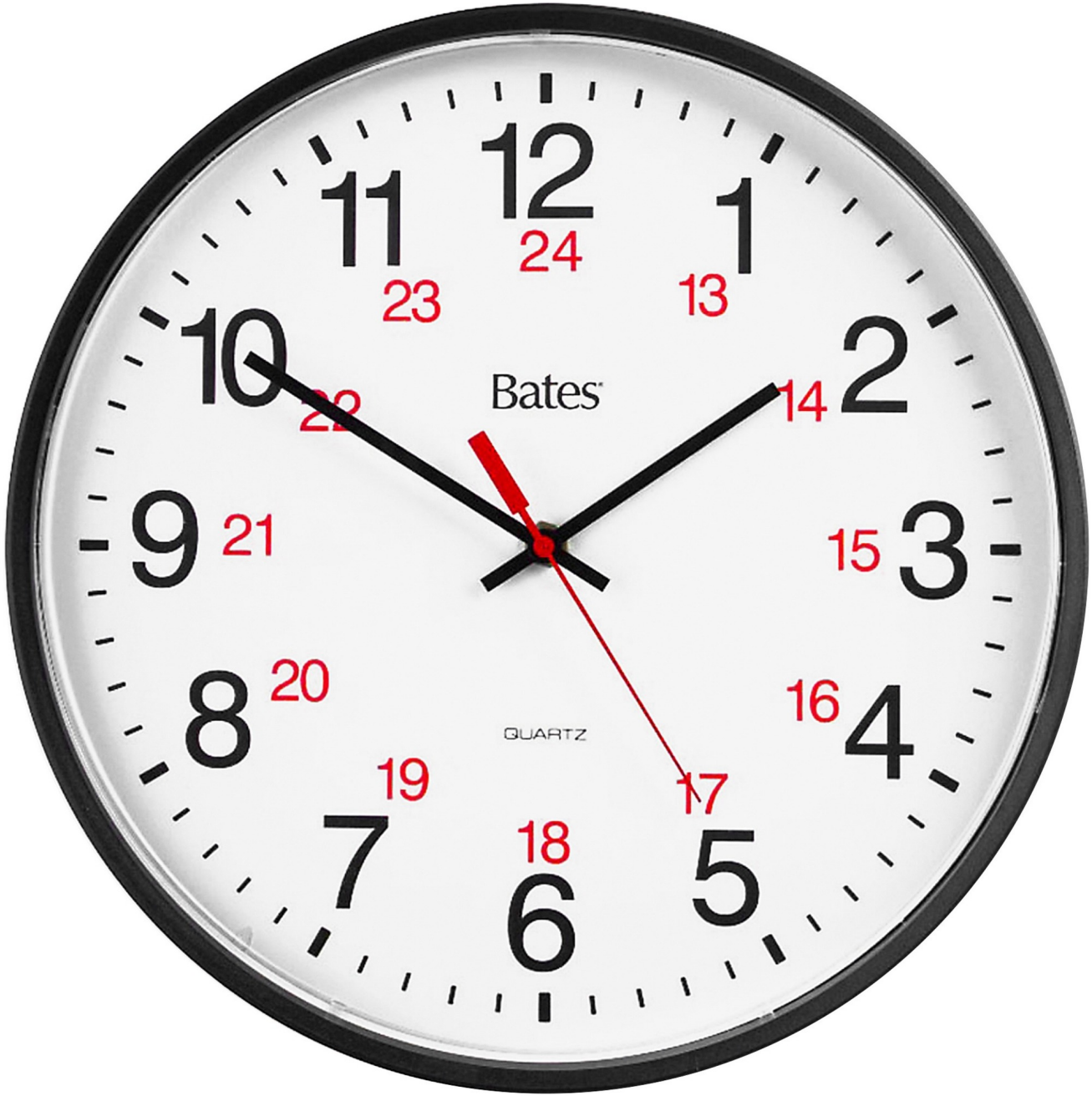

`plot(x)` Values of x in order.


`plot(x, y)` Values of x against y.


`hist(x)` Histogram of x.

Dates

See the **lubridate** package.


12
24
1
13
2
14
3
15
4
16
5
17
6
18
7
19
8
20
9
21
10
22
11
23
Bates
QUARTZ

functions_aula06.R | quadratica_2019.R

Source on Save | Run | Source

```r
quadratic <-function (a = 1, b = -2, c = 1){
            delta <- b^2-4*a*c
            preal <- -b/(2*a)
            pimag <- sqrt(abs(delta))/(2*a)
            if (delta == 0){
                print("raizes iguais")
                r1 <- preal
                r2 <- r1
            } else if (delta > 0){
                print("raizes reais e distintas")
                r1 <- preal + pimag
                r2 <- preal - pimag
            } else if (delta < 0){
                print("raizes complexas conjugadas")
                r1 <- complex(real = preal, imaginary = pimag)
                r2 <- complex(real = preal, imaginary = - pimag)
            }
            print (r1)
            print (r2)
}
```

```
> quadratic()
[1] "raizes iguais"
[1] 1
[1] 1
> quadratic(1,0,1)
[1] "raizes complexas conjugadas"
[1] 0+1i
[1] 0-1i
> quadratic(1,5,6)
[1] "raizes reais e distintas"
[1] -2
[1] -3
>
```

functions_aula06.R | quadratica_2019.R

Source on Save | Run | Source

```
1  quadratic <-function (a = 1, b = -2, c = 1){
2              delta <- b^2-4*a*c
3              preal <- -b/(2*a)
4              pimag <- sqrt(abs(delta))/(2*a)
5              if (delta == 0){
6                  print("raizes iguais")
7                  r1 <- preal
8                  r2 <- r1
9              } else if (delta > 0){
10                 print("raizes reais e distintas")
11                 r1 <- preal + pimag
12                 r2 <- preal - pimag
13             } else if (delta < 0){
14                 print("raizes complexas conjugadas")
15                 r1 <- complex(real = preal, imaginary = pimag)
16                 r2 <- complex(real = preal, imaginary = - pimag)
17             }
18 #              print (r1)
19 #              print (r2)
20             roots <- c(r1,r2)
21             return(roots)
22 }
23
```

```
> quadratic(1,5,6)
[1] "raizes reais e distintas"
[1] -2 -3
>
```

Problema 2

Implemente o método do ponto fixo em R

Ponto fixo

Os problemas de encontrar um ponto fixo de uma função e de encontrar uma de suas raízes são equivalentes, no seguinte sentido:

Dado um problema de determinação de raiz $f(p) = 0$, podemos definir funções g com um ponto fixo em p de diversas formas. Por exemplo, $g(x) = x - f(x)$ ou $g(x) = x + 3f(x)$.

Reciprocamente, se a função g tiver um ponto fixo em p, a função definida por $f(x) = x - g(x)$ terá um zero em p.

Método do Ponto Fixo - exemplo

A equação $x^3 + 4x^2 - 10 = 0$ tem uma única raiz em $[1, 2]$.

Existem várias maneiras pelas quais a equação pode ser manipulada para obter a forma de ponto fixo $x = g(x)$, como, por exemplo,

- $x = g_1(x) = x - x^3 - 4x^2 + 10$;
- $x = g_2(x) = \sqrt{\frac{10}{x} - 4x}$;
- $x = g_3(x) = \frac{1}{2}\sqrt{10 - x^3}$;
- $x = g_4(x) = \sqrt{\frac{10}{4+x}}$;
- $x = g_5(x) = x - \frac{x^3+4x^2-10}{3x^2+8x}$.

Algoritmo

Método do Ponto Fixo: dados uma aproximação inicial p_0, uma tolerância $TOL > 0$ e o número máximo de iterações N_0, devolve a solução aproximada p ou uma mensagem de erro.

Passo 1: Faça $k \leftarrow 1$.

Passo 2: Enquanto $k \leq N_0$, execute os passos 3 a 6:

Passo 3: Faça $p \leftarrow g(p_0)$.

Passo 4: Se $|p - p_0| < TOL$ ou $\frac{|p-p_0|}{|p|} < TOL$ ou $|f(p)| < TOL$, então devolva p como solução e pare.

Passo 5: Faça $k \leftarrow k + 1$.

Passo 6: Faça $p_0 \leftarrow p$.

Passo 7: Escreva "o método falhou após N_0 iterações" e pare.

Método do Ponto Fixo - exemplo

Com $p_0 = 1.5$, a tabela a seguir mostra os resultados da aplicação do Método do Ponto Fixo para as 5 opções de g. A raiz verdadeira é 1.365230013.

k	g_1	g_2	g_3	g_4	g_5
0	1.500	1.5000	1.500000000	1.500000000	1.500000000
1	-0.875	0.8165	1.286953768	1.348399725	1.373333333
2	6.732	2.9969	1.402540804	1.367376372	1.365262015
3	-469.700	$\sqrt{-8.65}$	1.345458374	1.364957015	1.365230014
4	1.03×10^8		1.375170253	1.365264748	1.365230013
5			1.360094193	1.365225594	
6			1.367846968	1.365230574	
7			1.363887004	1.365229942	
8			1.365916734	1.365230022	
9			1.364878217	1.365230012	
10			1.365410062	1.365230014	
15			1.365223680	1.365230013	
20			1.365230236		
25			1.365230006		
30			1.365230013		

Base R
Cheat Sheet

Getting Help

Accessing the help files

?mean
Get help of a particular function.
help.search('weighted mean')
Search the help files for a word or phrase.
help(package = 'dplyr')
Find help for a package.

More about an object

str(iris)
Get a summary of an object's structure.
class(iris)
Find the class an object belongs to.

Using Packages

install.packages('dplyr')
Download and install a package from CRAN.

library(dplyr)
Load the package into the session, making all its functions available to use.

dplyr::select
Use a particular function from a package.

data(iris)
Load a built-in dataset into the environment.

Working Directory

getwd()
Find the current working directory (where inputs are found and outputs are sent).

setwd('C://file/path')
Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

Vectors

Creating Vectors

`c(2, 4, 6)`	2 4 6	Join elements into a vector
`2:6`	2 3 4 5 6	An integer sequence
`seq(2, 3, by=0.5)`	2.0 2.5 3.0	A complex sequence
`rep(1:2, times=3)`	1 2 1 2 1 2	Repeat a vector
`rep(1:2, each=3)`	1 1 1 2 2 2	Repeat elements of a vector

Vector Functions

sort(x)
Return x sorted.
rev(x)
Return x reversed.
table(x)
See counts of values.
unique(x)
See unique values.

Selecting Vector Elements

By Position

`x[4]`	The fourth element.
`x[-4]`	All but the fourth.
`x[2:4]`	Elements two to four.
`x[-(2:4)]`	All elements except two to four.
`x[c(1, 5)]`	Elements one and five.

By Value

`x[x == 10]`	Elements which are equal to 10.
`x[x < 0]`	All elements less than zero.
`x[x %in% c(1, 2, 5)]`	Elements in the set 1, 2, 6.

Named Vectors

`x['apple']`	Element with name 'apple'.

Programming

For Loop

```
for (variable in sequence){
   Do something
}
```

Example

```
for (i in 1:4){
   j <- i + 10
   print(j)
}
```

While Loop

```
while (condition){
   Do something
}
```

Example

```
while (i < 5){
   print(i)
   i <- i + 1
}
```

If Statements

```
if (condition){
   Do something
} else {
   Do something different
}
```

Example

```
if (i > 3){
   print('Yes')
} else {
   print('No')
}
```

Functions

```
function_name <- function(var){
   Do something
   return(new_variable)
}
```

Example

```
square <- function(x){
   squared <- x*x
   return(squared)
}
```

Reading and Writing Data

Also see the **readr** package.

Input	Ouput	Description
`df <- read.table('file.txt')`	`write.table(df, 'file.txt')`	Read and write a delimited text file.
`df <- read.csv('file.csv')`	`write.csv(df, 'file.csv')`	Read and write a comma separated value file. This is a special case of read.table/ write.table.
`load('file.RData')`	`save(df, file = 'file.Rdata')`	Read and write an R data file, a file type special for R.

Conditions

`a == b`	Are equal	`a > b`	Greater than	`a >= b`	Greater than or equal to	`is.na(a)`	Is missing
`a != b`	Not equal	`a < b`	Less than	`a <= b`	Less than or equal to	`is.null(a)`	Is null

Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

`as.logical`	`TRUE, FALSE, TRUE`	Boolean values (TRUE or FALSE).
`as.numeric`	`1, 0, 1`	Integers or floating point numbers.
`as.character`	`'1', '0', '1'`	Character strings. Generally preferred to factors.
`as.factor`	`'1', '0', '1', levels: '1', '0'`	Character strings with preset levels. Needed for some statistical models.

Maths Functions

`log(x)`	Natural log.	`sum(x)`	Sum.
`exp(x)`	Exponential.	`mean(x)`	Mean.
`max(x)`	Largest element.	`median(x)`	Median.
`min(x)`	Smallest element.	`quantile(x)`	Percentage quantiles.
`round(x, n)`	Round to n decimal places.	`rank(x)`	Rank of elements.
`signif(x, n)`	Round to n significant figures.	`var(x)`	The variance.
`cor(x, y)`	Correlation.	`sd(x)`	The standard deviation.

Variable Assignment

```
> a <- 'apple'
> a
[1] 'apple'
```

The Environment

`ls()`	List all variables in the environment.
`rm(x)`	Remove x from the environment.
`rm(list = ls())`	Remove all variables from the environment.

You can use the environment panel in RStudio to browse variables in your environment.

Matrices

`m <- matrix(x, nrow = 3, ncol = 3)`
Create a matrix from x.


`m[2, ]` - Select a row


`m[ , 1]` - Select a column


`m[2, 3]` - Select an element

`t(m)`
Transpose

`m %*% n`
Matrix Multiplication

`solve(m, n)`
Find x in: m * x = n

Lists

`l <- list(x = 1:5, y = c('a', 'b'))`
A list is a collection of elements which can be of different types.

`l[[2]]`	`l[1]`	`l$x`	`l['y']`
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

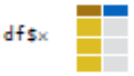
Also see the **dplyr** package.

Data Frames

`df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))`
A special case of a list where all elements are the same length.

x	y
1	a
2	b
3	c

List subsetting

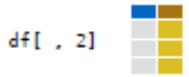

`df$x` `df[[2]]`

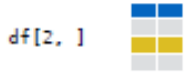
Understanding a data frame

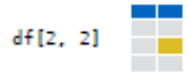
`View(df)` See the full data frame.

`head(df)` See the first 6 rows.

Matrix subsetting


`df[ , 2]`


`df[2, ]`


`df[2, 2]`

`nrow(df)` Number of rows.

`ncol(df)` Number of columns.

`dim(df)` Number of columns and rows.

`cbind` - Bind columns.

`rbind` - Bind rows.

Strings

Also see the **stringr** package.

`paste(x, y, sep = ' ')`	Join multiple vectors together.
`paste(x, collapse = ' ')`	Join elements of a vector together.
`grep(pattern, x)`	Find regular expression matches in x.
`gsub(pattern, replace, x)`	Replace matches in x with a string.
`toupper(x)`	Convert to uppercase.
`tolower(x)`	Convert to lowercase.
`nchar(x)`	Number of characters in a string.

Factors

`factor(x)`
Turn a vector into a factor. Can set the levels of the factor and the order.

`cut(x, breaks = 4)`
Turn a numeric vector into a factor by 'cutting' into sections.

Statistics

`lm(y ~ x, data=df)`
Linear model.

`glm(y ~ x, data=df)`
Generalised linear model.

`summary`
Get more detailed information out a model.

`t.test(x, y)`
Perform a t-test for difference between means.

`pairwise.t.test`
Perform a t-test for paired data.

`prop.test`
Test for a difference between proportions.

`aov`
Analysis of variance.

Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	`rnorm`	`dnorm`	`pnorm`	`qnorm`
Poisson	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** package.

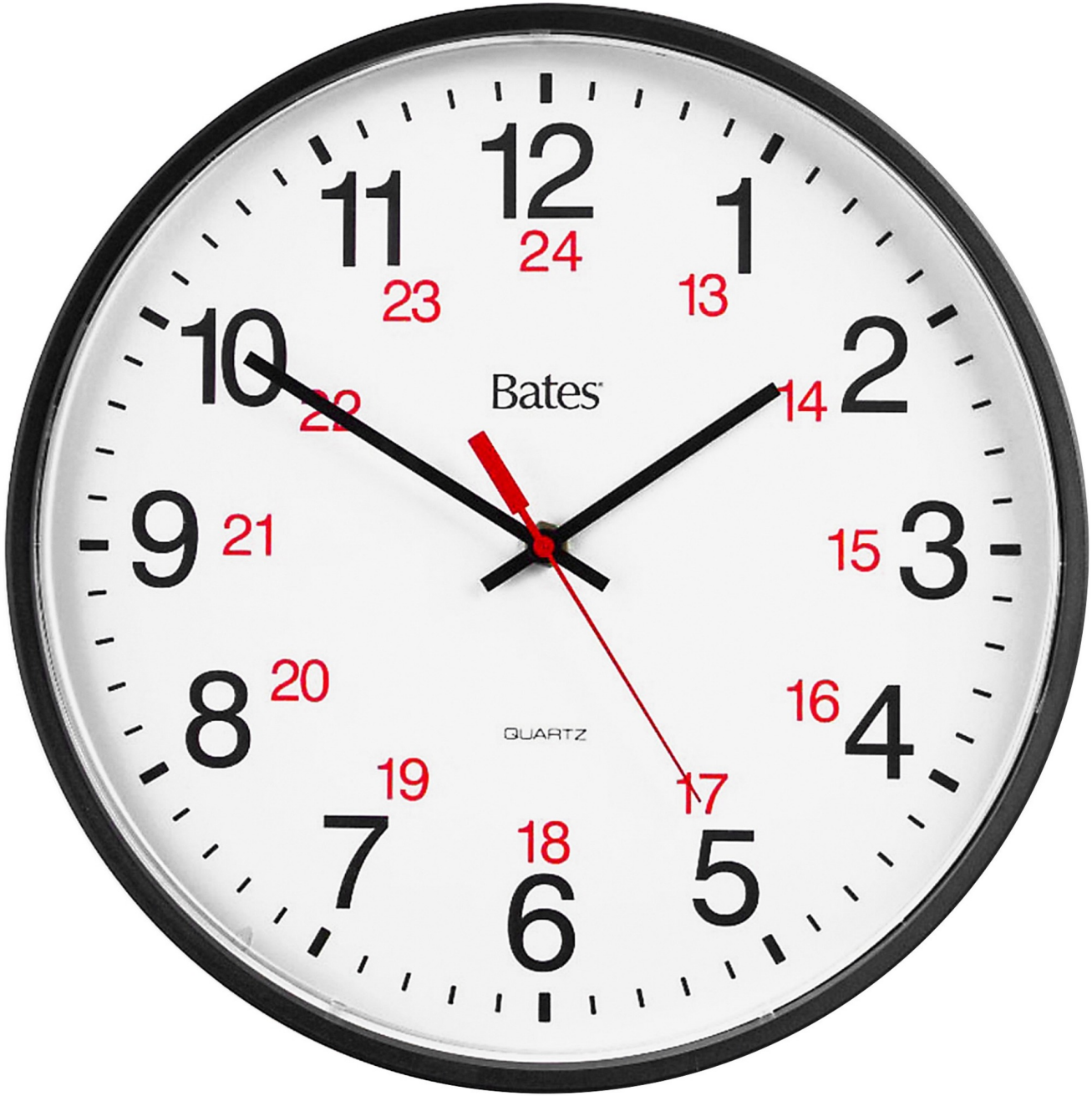

`plot(x)` Values of x in order.

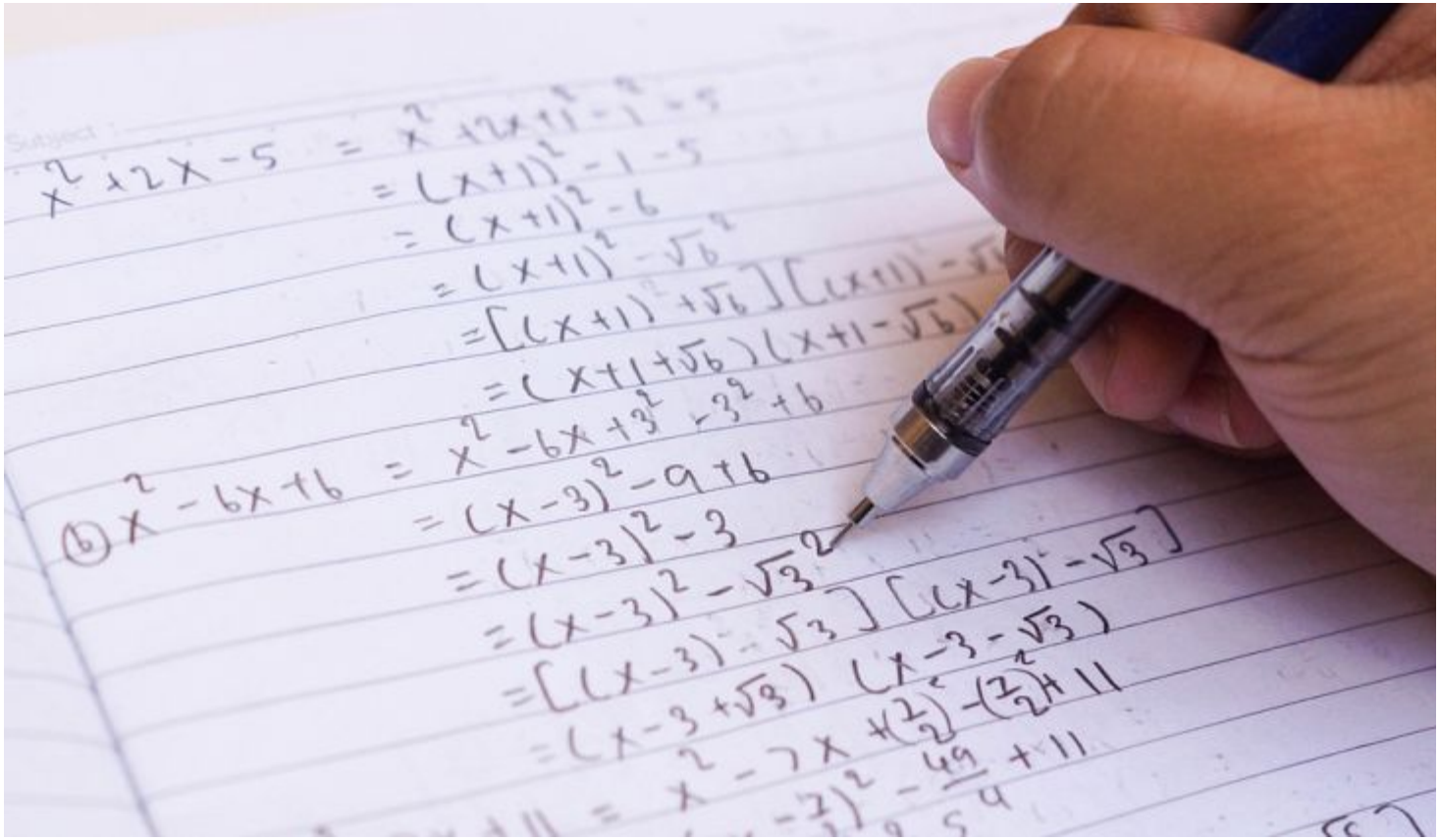

`plot(x, y)` Values of x against y.


`hist(x)` Histogram of x.

Dates

See the **lubridate** package.


12
24
1
13
2
14
3
15
4
16
5
17
6
18
7
19
8
20
9
21
10
22
11
23
Bates
QUARTZ



TESTEM SUAS ESCOLHAS NO LAB E VERIFIQUEM QUE RESOLVEM O PROBLEMA EXEMPLO

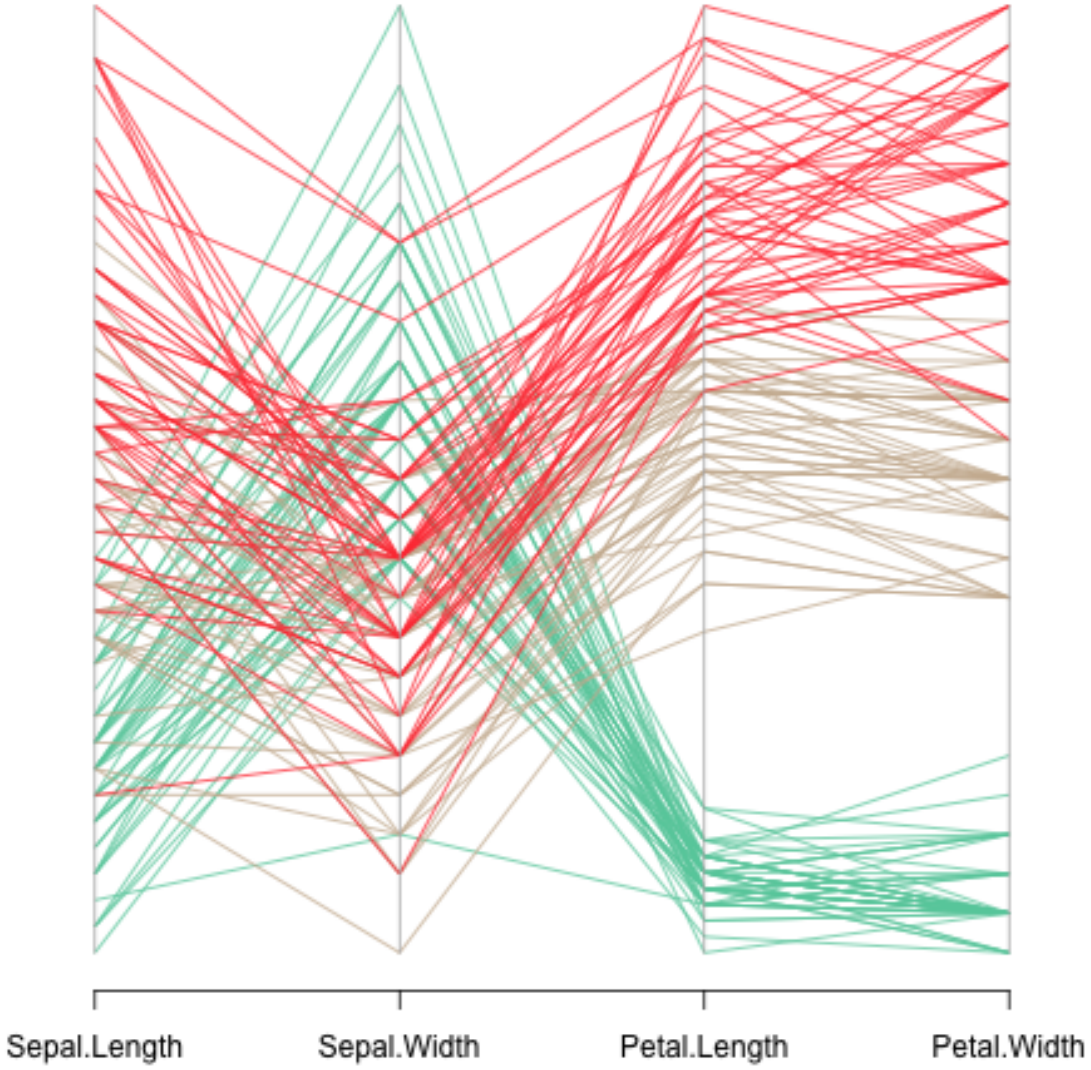
AVISOS:

a) Adiamento da 2ª prova de 21 de maio para 28 de maio (sala B-5, bloco B do IME)

b) 3ª prova mantida em 25 de junho caindo toda a matéria de R (conteúdos da 1ª e 2ª prova)

c) Duas aulas de preparação para o trabalho computacional (11 e 18 de junho)

d) Dia 04 de junho não haverá aula



https://www.r-graph-gallery.com/parallel-plot/