Introduction to R software

Bruna G. Palm

Fall, 2024

Important information

- Two labs covering content.
- The final scheduled lab is for questions.
- There will be a total of 2 assignments.

R is a computational system/environment for computing science and graphics

It is an implementation of the language of S programming. (There is a commercial implementation called S-PLUS)

R is free software, which means it is distributed for free and its source code is available

Website: `http://www.R-project.org`

R includes the base package or recommended packages with documentation

Additional packages can be installed with facility. E.g.,

```
> install.packages("betareg")
```

To use the package:

```
> library(betareg)
```

(or require(betareg))

To update packages:

```
> update.packages()
```

or

```
> update.packages(repos="http://cloud.r-project.org")
```

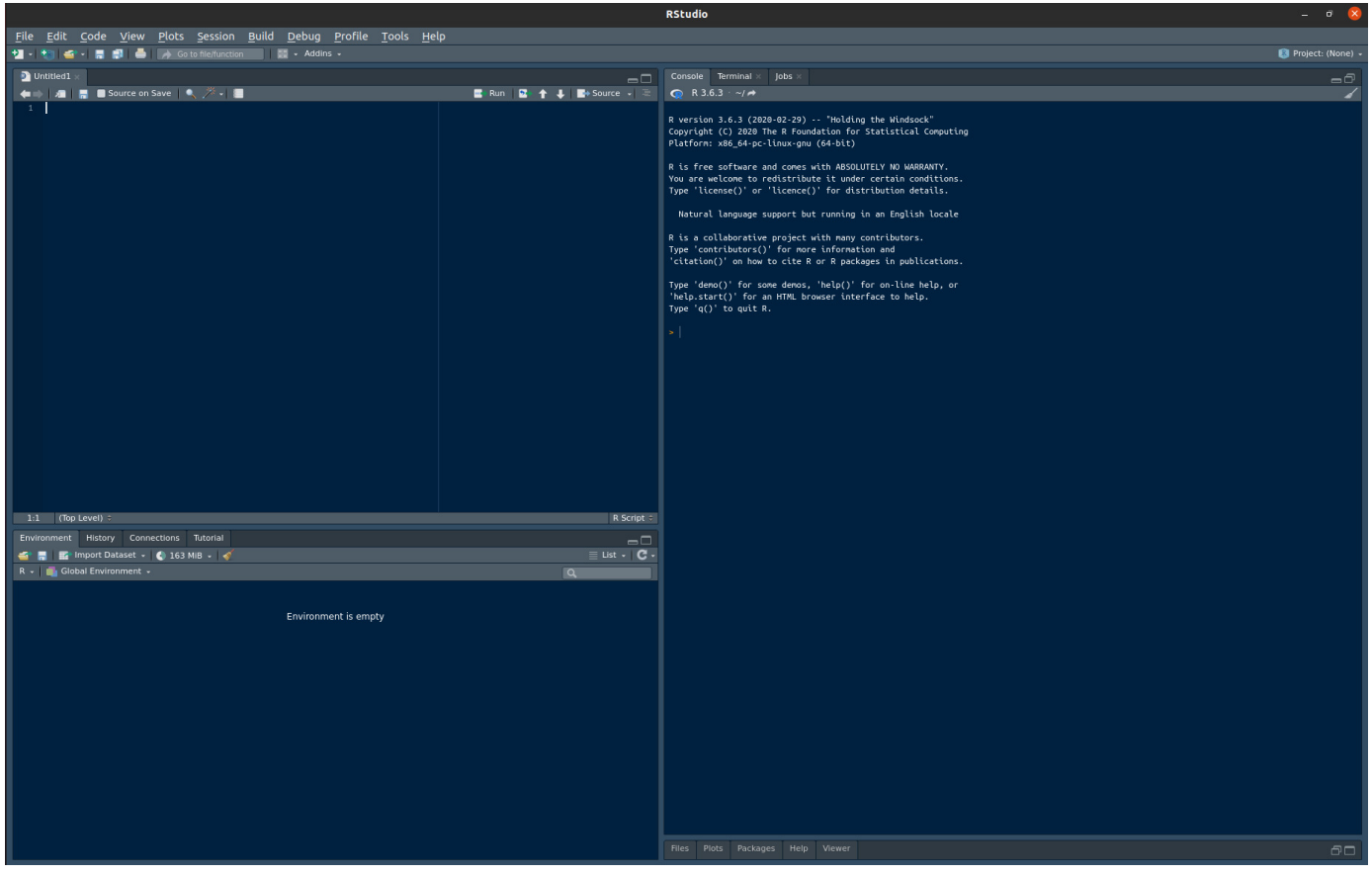
R is an interpreted language

After starting R you will find a prompt: >. From there, you can perform calculations interactively

IDE

There are IDEs that can be used

For example, RSTUDIO: `http://www.rstudio.com`



First evaluation:

```
> dnorm(0)
[1] 0.3989423
```

`[1]` indicates the first element of a vector

If the instruction spans more than one line, the prompt lines that follow change to +. E.g.,

```
> 1/sqrt(2*pi)*
+ exp(2)
[1] 2.947807
```

Designation: <- or =. E.g.,

```
> x = seq(0,3,0.5)
> x
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0
```

For information about a function, use `help` or `?`. E.g.,

```
> help(var)
```

or `?var`. Help page in a browser: `help.start()`. You may specify the browser: `help.start(browser="firefox")`

Comments to end of line: Use #.

To list variables in workspace: `ls()` or `objects()`

To remove a variable, use rm. E.g., `rm(x)`

Functions

We can extend the functionality of R by writing our own functions. The syntax is

```
function( LIST_OF_ARGUMENTS )
{
INSTRUCTIONS
return(VALUE)
}
```

Example: Function sum the values of n normally observed data that rolls n dice and returns the sum of values

```
DataSum = function(n=10){
k = sample(1:6, size=n, replace=TRUE)
return(sum(k))
}
```

Here, 10 is the default value of n

Functions

Use:

```
>DataSum(20)
[1] 81
>DataSum() # uses n=10
[1] 53
>DataSum(n=30)
[1] 92
```

To make changes to the role: `fix(data sum)`.

Functions

We can generalize the function to number of sides data different from 6:

```
OverallSumData = function(n=10, sides=6){
if(sides < 1) return(0)
k = sample(1:sides, size=n, replace=TRUE)
return((sum(k)))
}
```

Use:

```
OverallSumData(5,4)
OverallSumData(n=5, sides=4)
```

Matrices and Vectors

```
> x = 1:24 # vector
> A = matrix(1:24, nrow=4, ncol=6) # matrix
> A3 = array(1:24, c(3,4,2)) # array 3x4x2
> A
[.1] [.2] [.3] [.4] [.5] [.6]
[1,] 1 5 9 13 17 21
[2,] 2 6 10 14 18 22
[3,] 3 7 11 15 19 23
[4,] 4 8 12 16 20 24
> A[1,2]
[1] 5
> A[,2]
[1] 5 6 7 8
```

The filling of matrices is by columns. To fill in by rows, use `byrow=TRUE`.

Indexing starts at 1

Matrices and Vectors

Transpose of A: `t(A)`

Matrix product: `A %*% B`

$A'A$ `A t(A) %*% A` or `crossprod(A)`

Inverse of A: `solve(A)`

Number of rows and columns: Use `nrow` and `ncol`

Horizontal and vertical concatenation: Use `cbind` and `rbind`

Files

We can save the R code in a text file and run it from from R

For example, create a file called `prog.R` that contains the instructions

Then, in R,

```
source("prog.R")
```

Distributions

Indicate what you want to evaluate by the letter you put before the distribution name:

- `d`: density assessment
- `p`: distribution function evaluation
- `q`: quantile function evaluation
- `r`: random number generation

Distributions

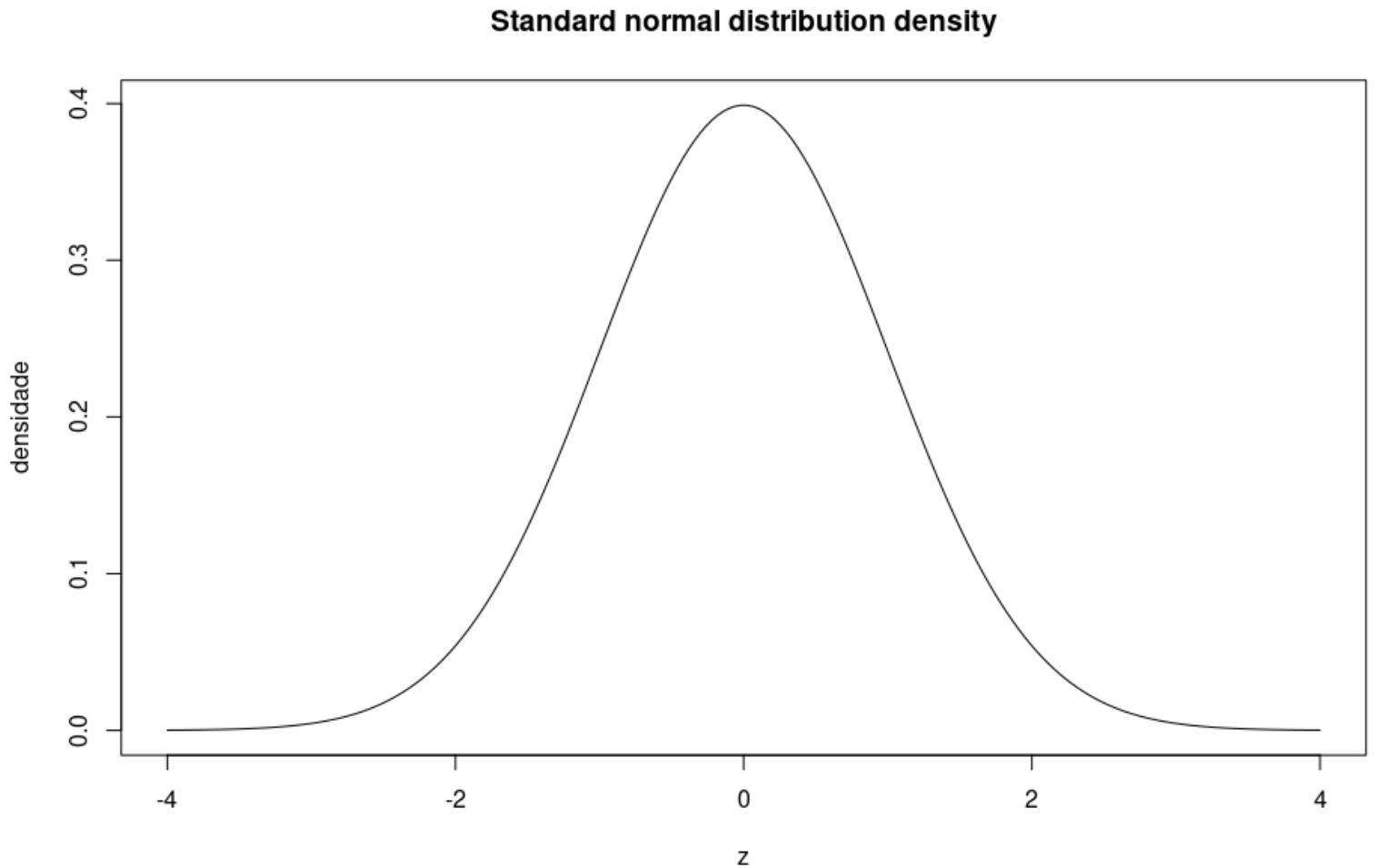
```
> x = rnorm(100000)
> mean(x)
[1] -0.005139324
> var(x)
[1] 1.004185
> summary(x)
Min. 1st Qu. Median Mean 3rd Qu. Max.
-4.3540 -0.68250 -0.008249 -0.005139 0.67160 4.4080
> hist(x)
> hist(x, prob=TRUE)
> lines(density(x), col="red")
> library(MASS)
> truehist(x)
> title("Histogram")
```

Distributions

Example of the standard normal distribution density:

```
> z = seq(from=-4, to=4, length=200)
> plot(z, dnorm(z), type="l", xlab="z",
ylab="densidade")
> title("Standard normal distribution density")
```

Distributions

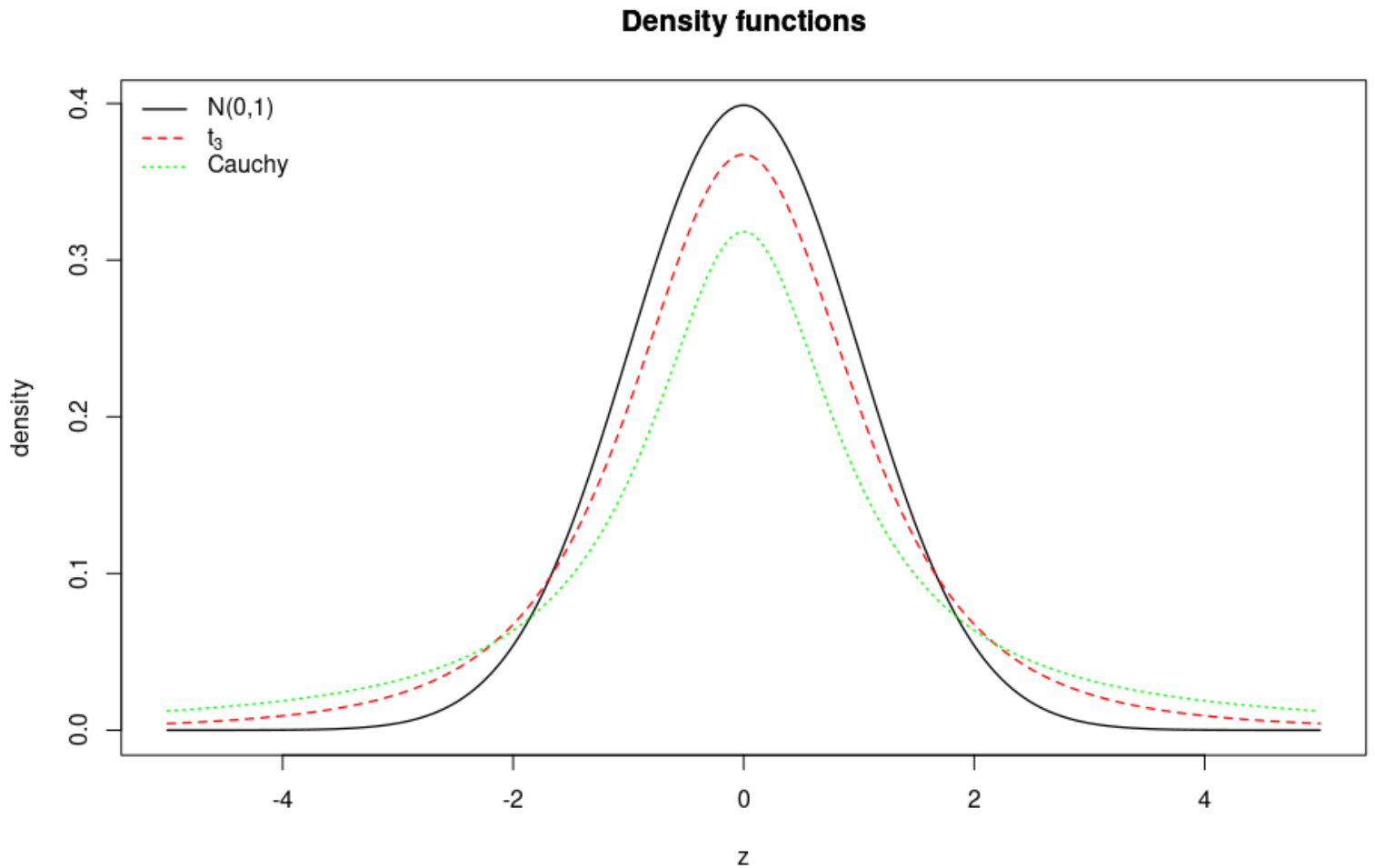


Distributions

Let's add three densities on the same graph: $N(0,1)$, t_3 and Cauchy

```
> z = seq(from=-5, to=5, length=200)
> zz = cbind(z,z,z)
> yy = cbind(dnorm(z), dt(z,3), dcauchy(z))
> matplot(zz, yy, type="l", lwd=1.3, xlab="z",
+ ylab="densities", col=c("black", "red", "green"))
> legend("topleft", lty=1:3, lwd=1.3, legend=c("N(0,1)",
+ expression(t[3]), "Cauchy"), col=c("black", "red",
+ "green"), bty="n")
> title("Density functions")
```

Distributions



Charts

To make a graphics panel, change the parameter that indicates how many graphics should appear on a page

For example, for make a 3×2 panel of graphs (3 rows and 2 columns) use:

```
par(mfrow=c(3,2))
```

To return to one chart per page:

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

Charts

To print a chart: `> dev.print()`.

To salvage a graphic for an EPS file, use a function `dev.copy2eps`

To save a graphic to a PDF file, use it `dev.copy2pdf`

Alternatively, use the postscript function. E.g.,

```
> postscript (file = "contornos.ps", paper = "letter",
+ horizontal = TRUE)
> contour (x, y, z)
> dev.off ()
```

Statements and Loops

Statements with if:

```
if(i == 1) {...}
```

Statements with while:

```
while(i <= 100) {...}
```

Statements with for:

```
for(i in 1:100) {...}
```

Statements and Loops

`break`: takes us out of the loop

`next`: takes us to the beginning of the loop

For conditional statements, use `if`, `else if`, `else`

There is also the `ifelse` function:

```
y = ifelse (x> 0, log (x), x)
```

Apply

We can often avoid using loops using the `apply` function

Use:

```
apply(array, 1 or 2, function)
```

1: operation in rows, 2: operation in columns

List

Lists can gave different types and dimensions We can create a list using the `list` function. E.g.,

```
> a = runif(50)
> b = norm(100)
> rnumbers = list(ru=a, rn=b)
> mean(rnumbers$ru)
[1] 0.4972335
> mean(rnumbers$rn)
[1] -0.02007798
> rnumbers$ru[1:5] # 5 first uniform numbers
[1] 0.3996454 0.5387299 0.6145015 0.3733817 0.5979650
```

Data Frames

A data frame is a list of variables, all from the same dimension, but not necessarily of the same type

Example:

```
> numbers = data.frame(a, b[1:50])
```

Data frames are typically created when reading data from a external file using the `read.table` function. E.g.,

```
> y = read.table("data.txt", header=TRUE)
```

If the lines have names, use the `row.names=TRUE` option.

IMPORTANT: Data frames can be indexed as lists or as matrices.

Seed

To set a seed, use the `set.seed` function. E.g.,

```
> set.seed(1970)
```

Nonlinear Optimization

Use the `optim` function

By default, R does minimization. To maximize, multiply the function by -1 or use the option `control=list(fnscale=-1)`

Nonlinear Optimization

Available optimization methods:

- “BFGS”
- “CG” (Conjugate gradient)
- “L-BFGS-B” (BFGS with restrictions)
- “Nelder-Mead” (simplex)
- “SANN” (simulated annealing)
- “Brendt” (uni-dimensional)

Nonlinear Optimization

Return of function:

`convergence:`

`0: there was convergence`
`1: maximum number of iterations reached`
`10: there was degradation of the simplex method`
`51 and 52: there were problems with the L-BFGS-B`

`message`: a string with information
`par`: the maximizes found
`value`: the value of the function in `par`
`hessian`: hessian estimate

Nonlinear Optimization

Example: Minimizing the Rosenbrock function:

$$F(\alpha, \beta) = 100 \times (\beta - \alpha^2)^2 + (1 - \alpha)^2$$

Function:

```
> fr <- function(x) {
+ x1 <- x[1]
+ x2 <- x[2]
+ 100 * (x2 - x1 * x1)^2 + (1 - x1)^2
+ }
```

Nonlinear Optimization

Gradient:

```
> grr <- function(x) {
+ x1 <- x[1]
+ x2 <- x[2]
+ c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
+ 200 * (x2 - x1 * x1))
+ }
```

Nonlinear Optimization

Using Nelder-Mead:

```
> optim(c(0, 0), fr)
$par
[1] 0.999956370130 0.999908469592
$value
[1] 3.72905186712e-09
$counts
function gradient
169 NA
$convergence
[1] 0
$message
NULL
```

Nonlinear Optimization

Using BFGS with numerical gradient:

```
> optim(c(0, 0), fr, method="BFGS")
$par
[1] 0.999800047990 0.999600132168
$value
[1] 3.9980807715e-08
$counts
function gradient
63 26
$convergence
[1] 0
$message
NULL
```

Nonlinear Optimization

Using BFGS with analytical gradient:

```
> optim(c(0, 0), fr, grr, method="BFGS")
$par
[1] 1.00000000122 1.00000000235
$value
[1] 2.31196631776e-18
$counts
function gradient
57 28
$convergence
[1] 0
$message
NULL
```