

# Introducing the R statistical language

Matteo Fasiolo

# R Programming

R is a high-level **statistical programming language**.

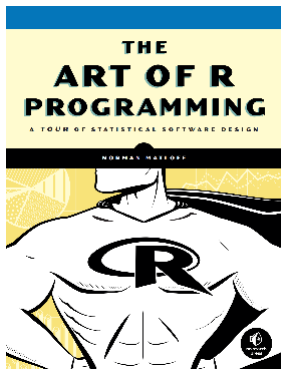
R is very efficient at

- ▶ vectors and matrices operations
- ▶ large datasets processing
- ▶ data visualization
- ▶ statistical/machine-learning modelling

One of the most popular statistical programming languages:

- ▶ you can find online solutions to your programming questions.
- ▶ Stackoverflow
- ▶ CrossValidated

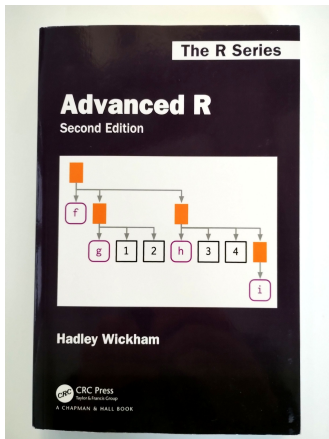
**The Art of R Programming: a tour of statistical software design.**



Ebook available via the [University library](#).

If you want something more in depth on the language itself:

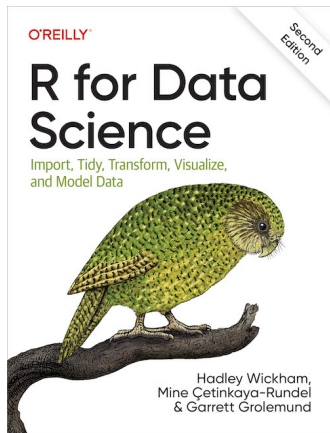
## Advanced R (second edition)



Ebook freely available [here](#). There is also a book with solutions to the exercises.

For something more focussed on data analysis in R:

## R for Data Science (second edition)



Ebook freely available [here](#).

Note that this relies on a set of R packages (the Tidyverse) that we will not use.

# Course objectives

Understand essential features of R programming.

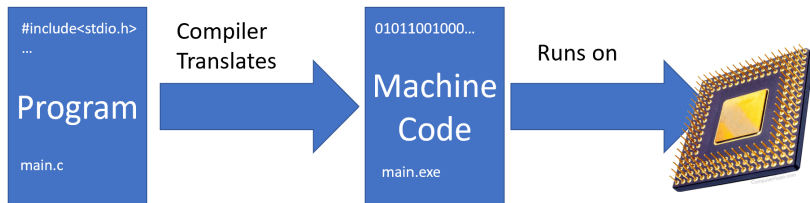
Be able to write and test basic **statistical algorithms** in R, with appropriate coding paradigms.

Understand the **differences between C and R**, and be able to decide which one to use when facing a task.

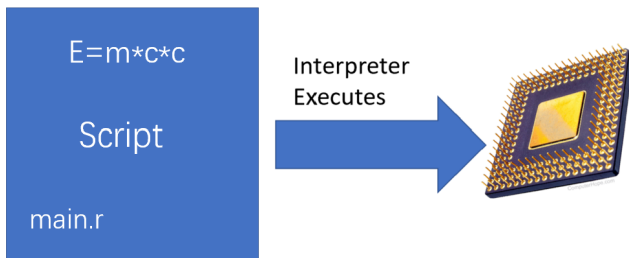
Be able to code a basic data science project using R by calling existing libraries.

# R is an Interpreted Language

Compiled programming language (e.g., C/C++)



Interpreted programming language (e.g., R/Python)



# Interpreted Languages

Interpreted languages do **not** need compilation.

The script is sent to a software called “interpreter” and is executed by it.

No executable file is produced (there is no app!).

The interpreter executes your code file **line by line**, and **you can even stop your program and make changes**.

It is impossible to do so in compiled languages. Once your program has been compiled and started running, you cannot stop it and modify the code.



# Examples of Interpreted Languages

Most websites are written by interpreted languages:

- ▶ HTML and Javascript are both interpreted languages.
- ▶ Your browsers (e.g. Edge/Chrome) are interpreters.
- ▶ They download, interpret the program (webpage source code) and render the outcome to the screen.

Most data science languages are interpreted languages:

- ▶ MATLAB/Python/R.
- ▶ Interpreted language allows users to stop the execution, inspect intermediate outcomes and make necessary changes.

## Pros

- ▶ No compilation step needed. Runs immediately.
- ▶ Flexible coding. No need to write the whole program in one go. You can delay the programming until you see the earlier execution results.

## Cons

- ▶ Slower than compiled language, code requires interpretation.
- ▶ No executable is produced. To run your code, your users must have **your code** and **install the interpreter**.
- ▶ Some interpreters, like MATLAB, are not free.

## Pro or Con:

- ▶ The source code is visible to the user.

Languages such as Julia have a JIT compiler and might offer the best of both worlds.

# Interpreted Languages are Ideal for Data Science Projects

Common data science project workflow:

1. Parse/Load the dataset from file.
2. Inspect the dataset **interactively**:
  - ▶ visualize some basic facts about your dataset.
  - ▶ determine what analysis you would like to run.
3. Code the algorithm
  - ▶ inspect the outcome of the algorithm
  - ▶ determine how to visualize the outcome.
4. Code the visualization part

**Key point:** often we do not know what we want to do in advance.

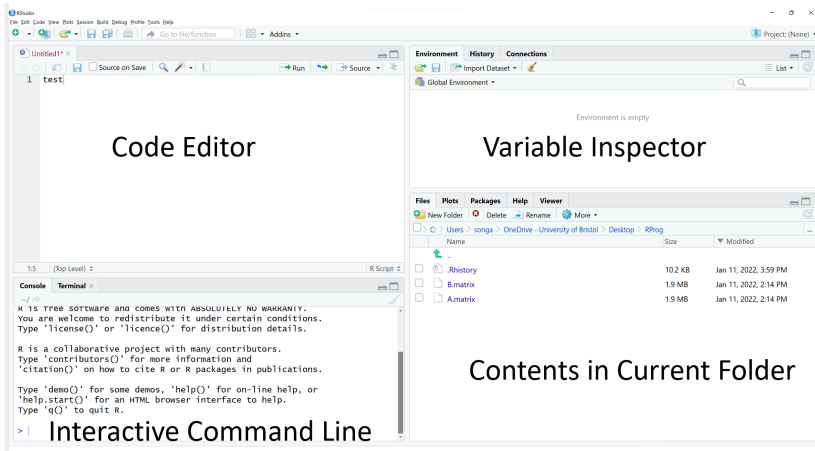
Interpreted Language allows you to cut your workflow into pieces, and program them adaptively, not in one go.



R allows your code to be adaptive, based on previous execution results.

# RStudio: An R Development Environment

Instead of using VSCode, we will use RStudio as the development environment for R programming.



**NOTE** Rstudio != R in same way as VSCode != C++.

Note that R comes with a set of **recommended packages**.

You can install many more packages from the Comprehensive R Archive Network (CRAN).

On the 7th of November 2024, there were 21624 packages on CRAN.

Tutorial on RStudio.

# Scalar Variables in R

To create a scalar variable in R:

```
a <- 10  
# Comments in R starts with #
```

Unlike C, you do not to specify the variable type. **R will guess the variable type.**

The assignment operator in R is <=.

a = 10 also works, but we recommend you use <=.

You can inspect a's value by typing a in the command line.

```
a
```

```
[1] 10
```

# Data Types in R

R has 5 basic data types: numerical (double), integer, character, logical and complex.

```
> a <- 10
> typeof(a) # what is the type of a?
[1] "double"
```

```
> a <- 10L # Appending "L" indicates an integer
> typeof(a)
[1] "integer"
```

```
> a <- TRUE # TRUE or FALSE
> typeof(a)
[1] "logical"
```

```
a <- "hello world!"
> typeof(a)
[1] "character"
```



Arithmetic and Logical/Relational operators are mostly the same as in C (see Sec 7.2 in ART). There are a few differences:

%% modular arithmetic

```
10%%3  
[1] 1
```

%% integer division

```
10%/%3  
[1] 3
```

Standard division

```
10/3  
[1] 3.333333
```

^ Exponentiation

```
2^10 # 2 to the 10th power  
[1] 1024
```

# Vectors in R

You can create and manipulate a vector easily.

```
v <- c(1,2,3,4)
v[1]
```

```
[1] 1
```

Note: R **uses 1-based index**, different from C and python.

Here c stands for “combine” or “concatenate”. Example:

```
v2 <- c(5,6,7,8)
c(v, v2)
```

```
[1] 1 2 3 4 5 6 7 8
```

c works also with R lists.

# Flow Control in R

If and If-Else in R is exactly the same as in C (See ART Sec 7.1).

```
a <- 1
if (a == 1){
  print("a is one!")
}
[1] "a is one!"
```

```
a <- 2
if (a == 1){
  print("a is one!")
} else {
  print("a is NOT one!")
}
[1] "a is NOT one!"
```

## If-Else Ladder

```
a <- 3
if (a == 1) {
  print("a is one!");
} else if (a == 2) {
  print("a is two!");
} else if (a == 3) {
  print("a is three!");
} else {
  print("I do not know!");
}
[1] "a is three!"
```

Same as in C but printf -> print.

## While Loop

While loop is exactly the same as in C (See ART Sec 7.1).

```
a <- 10
while(a>0){
  if(a<5){
    break
  }
  a <- a - 1 # a-- will not work!
}
a
[1] 4
```

break works in the same way too.

To skip an iteration in R loop, use next.

There is no do-while loop in R.

## For Loop

For loop in R is slightly different from C (See ART Sec 7.1):

```
for (i in 1:10){  
  # i takes 1 in the first iteration,  
  # i takes 2 in the second iteration ...  
  print(i)  
}  
[1] 1  
[1] 2  
[1] 3  
[1] 4  
[1] 5  
[1] 6  
[1] 7  
[1] 8  
[1] 9  
[1] 10
```

Step by step:

```
a <- 1:10
```

```
a
```

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

Then in `for (i in a)` will take each value in `a`.

Another example:

```
letters[1:3]
```

```
[1] "a" "b" "c"
```

```
for (i in letters[1:3]){  
  cat(i)  
}
```

abc

You can get nice output by using paste:

```
paste("a", "b")
```

```
[1] "a b"
```

```
paste(letters[1], letters[2])
```

```
[1] "a b"
```

So

```
for (i in letters[1:3]){  
  cat(paste(i, "\n"))  
}
```

```
a  
b  
c
```



# Built-in Functions

There are many built-in statistical/mathematical functions in R.

We can call them directly, without loading any library.

For example, to find the absolute value of `a`, we can

```
> a <- -10  
> abs(a)  
[1] 10
```

If you want help on the usage of `abs`, you can simply type

```
> ?abs
```

Help should show up on the right pane in Rstudio.

# Write Your Own Functions

You can write your own function using the following syntax:

```
smaller_than_10 <- function(n){  
  if(n<10){  
    return(TRUE)  
  }else{  
    return(FALSE)  
  }  
}
```

Function name followed by <- function(argument list).

You can call a function in the same way as in C:

```
> smaller_than_10(12)  
[1] FALSE  
> smaller_than_10(0)  
[1] TRUE
```

Functions behave as if inputs are **passed by value**:

```
dosomething <- function(v){  
  v[1] <- -10  
  return(v)  
}
```

```
a <- c(1,2,3,4)
```

```
dosomething(a)  
[1] -10  2  3  4
```

```
a  
[1] 1 2 3 4
```

```
v  
# Error: object 'v' not found  
# v is a local variable, not visible outside of function
```

Truth is that R uses **copy-on-modify**: A copy is made only when objects are modified.

Example:

```
x <- c(1, 2)
y <- x
```

Is y a copy of x?

```
tracemem(x)
```

```
[1] "<0x55f01240d998>"
```

```
tracemem(y)
```

```
[1] "<0x55f01240d998>"
```

No, but it becomes a copy as soon as you modify it:

```
y[1] <- 3
```

```
tracemem[0x55f01240d998 -> 0x55f012967ff8]: eval eval eval
```

The same happens with functions:

```
a <- c(1,2,3,4)
tracemem(a)
```

```
[1] "<0x55f01140c828>"
```

```
dosomething <- function(v){
  tracemem(v)
  v[1] <- -10
  return(v)
}
z <- dosomething(a)
```

```
tracemem[0x55f01140c828 -> 0x55f0114ac9e8]: dosomething evaluated
```

You can work routinely in R without knowing about this.

For an example where this matters, see the bonus slides at the end.

# Conclusion

1. R is an interpreted, high-level, statistical language.
  - ▶ Pros and Cons
2. Interpreted language does not need compilation and your code can be modified when your program is running.
3. R's **scalar syntax** is very similar to C but:
  - ▶ Assignment uses `<-`.
  - ▶ No need to declare a variable before assignment.
  - ▶ Comments start with `#`.
  - ▶ No need to add `;` at the end of each statement.

We will see that vector programming in R can be very different from what we have seen in C.

# Homework (Pre-sessional work)

## 1. Download R

- ▶ <https://www.stats.bris.ac.uk/R/>

## 2. Install and Launch RStudio

- ▶ <https://www.rstudio.com/products/rstudio/download/#download>

## 3. Try code blocks in the slides

- ▶ Write code in the code editor and press ctrl+enter to execute the code line by line.
- ▶ Create R scripts and save them.

If you are on university PCs, you can skip the first two steps.

## Homework (Submit)

Write a program that determines the number of primes smaller or equal than a natural number  $n$ ,  $n \geq 2$ .

Recall:

- ▶ A factor is an integer that divides exactly into a whole number without a remainder. E.g., 3 is a factor of 12.
- ▶ A prime has only 2 factors, 1 and itself.

**Pseudo Code**, suppose that  $n$  is the number of interest:

Loop over  $i$  going from 2 to  $n$  (1 is not a prime!)

1. Loop over  $j$  going from 1 to  $i$ 
  - ▶ If  $j$  is factor of  $i$  add it to the count of factors of  $i$
2. If  $i$  has only two factors, it is a prime and it should be added to the number of primes  $\leq n$



# Homework (Submit)

You should:

1. Translate above pseudo code into R code
  - ▶ Use only for() loops.
  - ▶ Write your code in the code editor and test it.
  - ▶ After the execution, check the “environment pane” on the top right corner of Rstudio:
    - ▶ What are the variables?
    - ▶ Why do they have the value they hold?
2. What is the computational complexity of our code?
  - ▶ Hint: count how many loop iterations will be executed when the program runs.

# Homework

## 3. Time the execution of your code using `Sys.time()`

```
start_time <- Sys.time()
#your code here
end_time <- Sys.time()
end_time - start_time

# Time difference of 0.0009999275 secs
```

Set `n <- 5000` and time the execution (select all and ctrl + enter).

How long does it take?

Predict how long it will take when setting `n <- 10000` before running the code.

Validate your prediction by actually running the code.

## Homework (Challenge)

4. Are you able to wrap your R code into a function?
5. Try to make your code faster using a smarter algorithm:
  - ▶ Hint: in loop over  $j$ , if  $i$  already has 3 factors then you already know that it's not a prime.
  - ▶ Hint: prime numbers are all odd (except for 2).
  - ▶ Compare the smart algorithm with the original version in terms of correctness and speed.
6. Write the same prime number counting program in C, compile and run it:
  - ▶ When setting  $n = 5000$ , which programming language is faster? faster by how much?
  - ▶ Use the skeleton C code in `Lab_Primes_template.c`.

## BONUS SLIDES

Here we are creating a vector with 10 million elements and we are setting the  $ii$ -th element to 1 at each iteration:

```
x <- 1:1e7
n <- 500

tic <- Sys.time()
for(ii in 1:n){
  x[ii] <- 1
}
( Sys.time() - tic ) # 0.1 seconds on my computer
```

Time difference of 0.1573076 secs

the function `Sys.time` returns the current time.

Here we do the same but using a function

```
dosomething <- function(v, ii){  
  v[ii] <- 1  
  return(v)  
}  
  
tic <- Sys.time()  
for(ii in 1:n){  
  x <- dosomething(v = x, ii)  
}  
( Sys.time() - tic ) # 30 seconds on my computer,  
                     # 300 times slower
```

Here the whole vector x gets copied each time we run `v[ii] <- 1`.