Introduction to R

an interpreted language

Song Liu (song.liu@bristol.ac.uk)

GA 18, Fry Building,

Microsoft Teams (search "song liu").

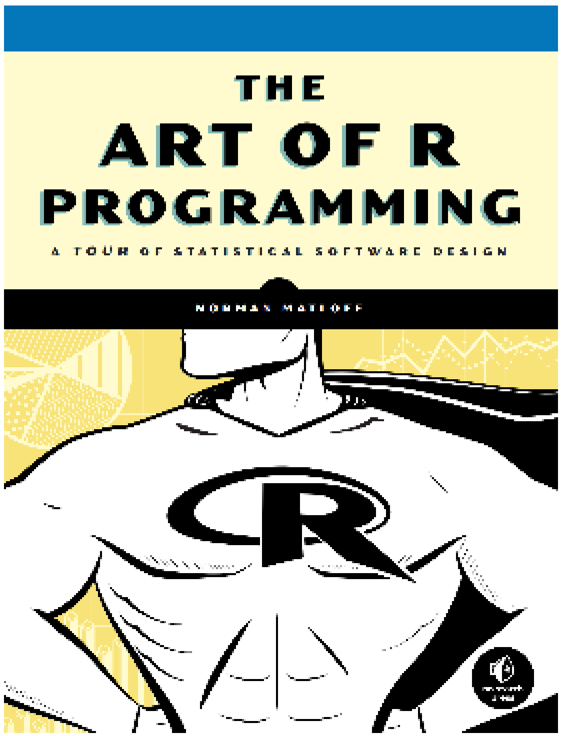
R Programming

- R is a high-level **statistical programming language**.
- R is very efficient at
  - vectors and matrices operations
  - large datasets processing
  - data visualization
- One of the most popular data mining programming language.
  - means you can find online posts/communities that solves your programming questions.
  - Stackoverflow.

R Book

The Art of R Programming:

a tour of statistical software design.



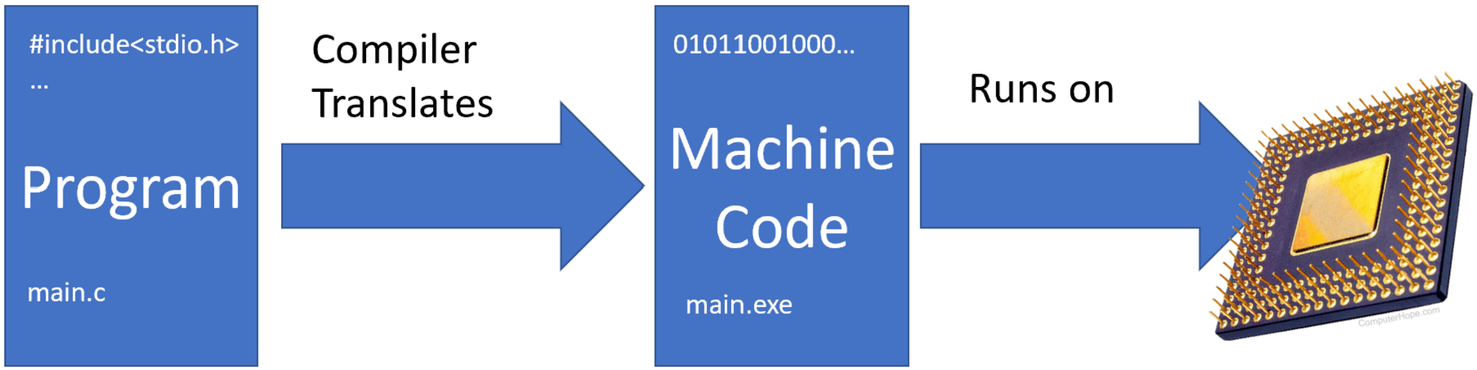
You can find the PDF version of this book (please google).

Key Objectives

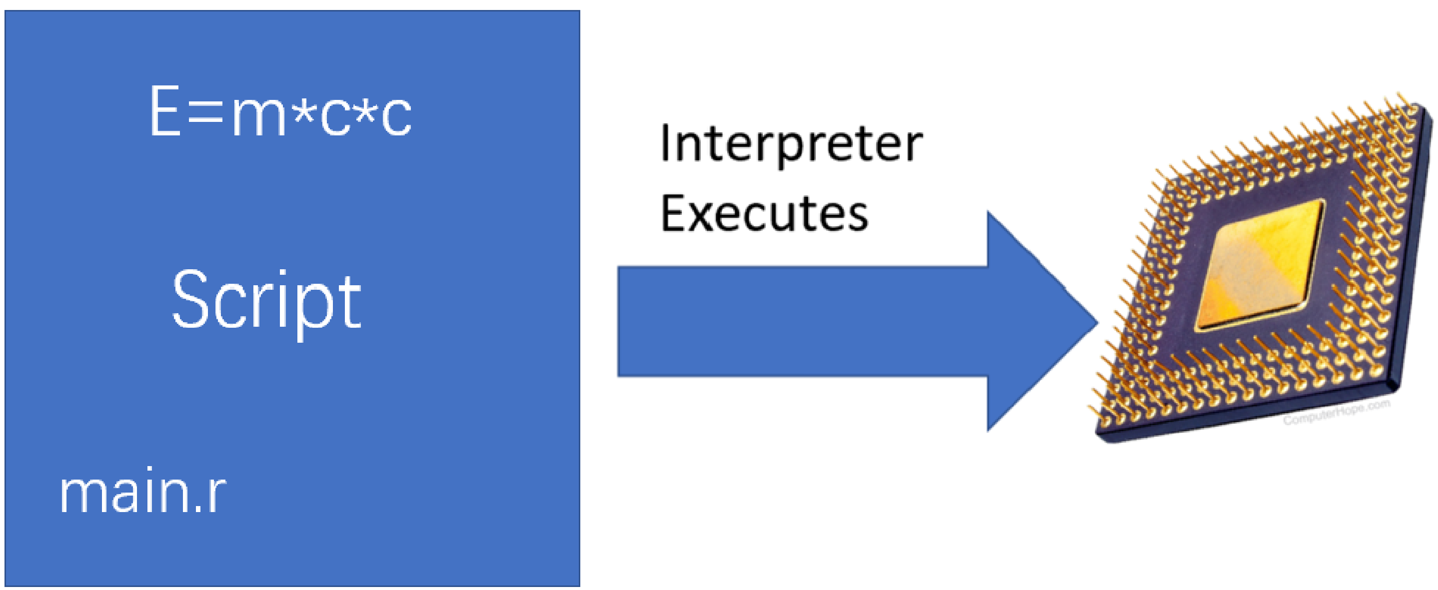
- Understand essential features of R programming.
- Be able to write and test basic **statistical algorithms** in R, with appropriate coding paradigms.
- Understand the **differences between C and R**, and be able to decide which one to use when facing a task.
- Be able to code a basic data science project using R by invoking existing statistics/data science libraries.

R is an Interpreted Language

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



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



Interpreted Language

- Interpreted language does **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 Language

- 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 and Cons

- 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, is not free.
    - Not really a ``con'' in many cases.

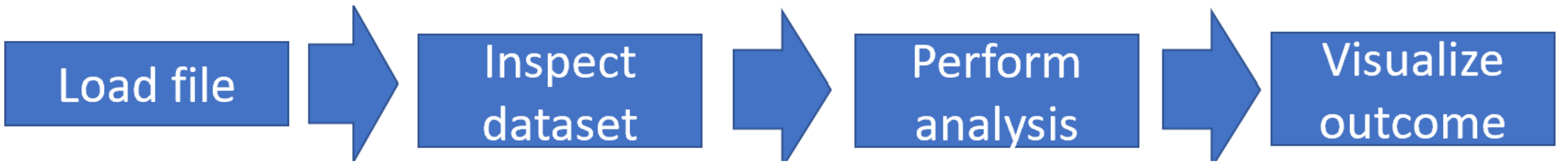
Interpreted Language is Ideal for Data Science Projects

Common data science project workflow:

1. Parse/Load the dataset from file.
2. Inspect the dataset.
    - 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

Interpreted Language is Ideal for Data Science Projects

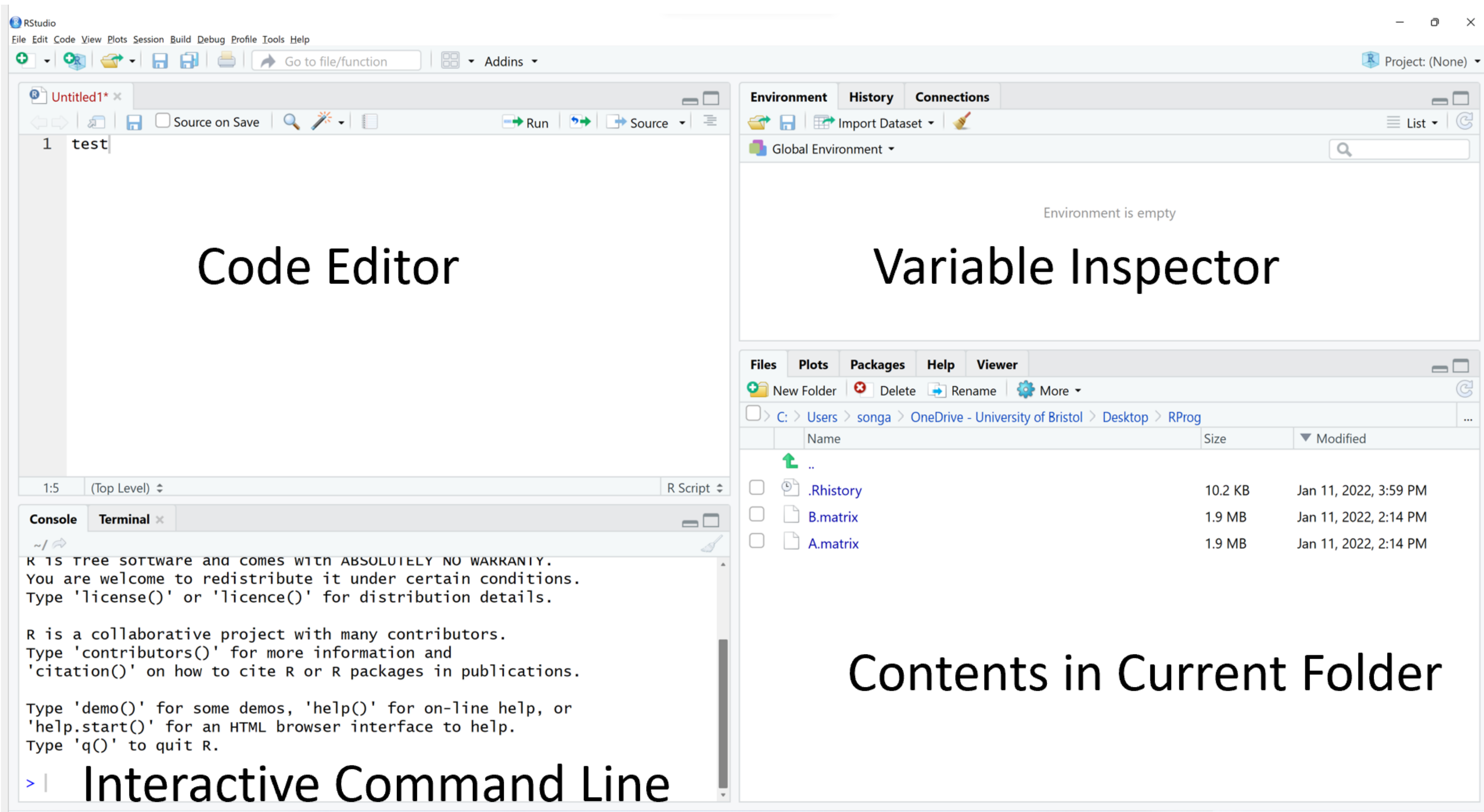
- Interpreted Language allows you to cut your workflow into pieces, and program them adaptively, not in one go.



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

RStudio: The R Development Environment

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



RStudio: The R Development Environment

Tutorial on RStudio.

Scalar Variables in R

To create a scalar variable in R:

```
a <- 10
#Now a is a scalar variable stores the value 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 `<-`. Although it also recognize `a=10`, 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 data types: numerical (double), integer, character, logical and complex.

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

```
> a <- 10L # You need to append "L", otherwise,
# R thinks it is a double.
> typeof(a)
[1] "integer"
```

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

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

Arithmetics in R

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
```

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

Arithmetics in R

`^` Exponentiation

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

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!"
```

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 the one in C (See ART Sec 7.1). To loop over a sequence of numbers, we can do

```
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
```

Built-in Functions

There are many built-in statistical/mathematical functions in R we can call directly. 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.

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 appears at the beginning 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
```

Vectors in R

There is no "array" in R. However, you can create and manipulate a vector easily in R.

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

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

- Here `c` stands for "combine" or "concatenate".

Vectors are Always Passed by Value

Input variables in R functions are **passed by value**!

```
a <- c(1,2,3,4)
dosomething <- function(v){
  v[1] = -10
  return(v)
}
dosomething(a)
[1] -10   2   3   4
a
[1] 1 2 3 4
v
# v is a local variable
# and is not visible from outside of the function "t"
# If you want to inspect v's value, R interpreter will
# return the following error:
Error: object 'v' not found
```

Conclusion

1. R is a 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.
    - Assignment uses `<-`.
    - No need to declare a variable before assignment.
    - Comments start with `#`.
    - No need to add `;` at the end of each statement.

However, as we will see in the next week, vector programming in R can be very different from what we have seen in C.

Homework (Pre-sessional work)

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

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.
- Watch the lecture video for demonstrations.

Homework

Write a program that determines the number of primes smaller or equal than a natural number `n, n>=2`.

Pseudo Code (fill out the blanks):

```
given a natural number n

set num_primes to 0

For i = 2 to n
  set num_factors to 0
  For j = 1 to i
     _______
     _______
  If num_factors equals to 2
    print i
    increase num_primes by 1
```

Homework (Submit)

2. Translate above pseudo code into R code
    - Write your code in the code editor and test it.
    - After the execution, check the "environment pane" on the top right corner.
        - What are the variables?
        - Why do they have the value they hold?
3. What is the computational complexity of our prime finding 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. Write the same prime number counting program in C, compile and run it.
    - When setting `n = 4000`, which programming language is faster? faster by how much?
    - Use the provided skeleton C code.