Introduction to R programming – a SciLife Lab course

Marcin Kierczak with Thomas Källman (labs)

31 August 2016

R – what is it really?

- a programming language,
- a programming platform (= environment + interpreter),
- a software project driven by the core team and the community.

And more:

- a very powerful tool for statistical computing,
- a very powerful computational tool in general. . . ,
- a catalyst between an idea and its presentation.

What R is not?

- a tool to replace a statistician,
- the very best programming language,
- the most elegant programming solution,
- the most efficient programming language.

Figure 1:

A brief history of R

- conceived c.a. 1992 by Robert Gentleman and Ross Ihaka (R&R) at the University of Auckland, NZ – a tool for teaching statistics,

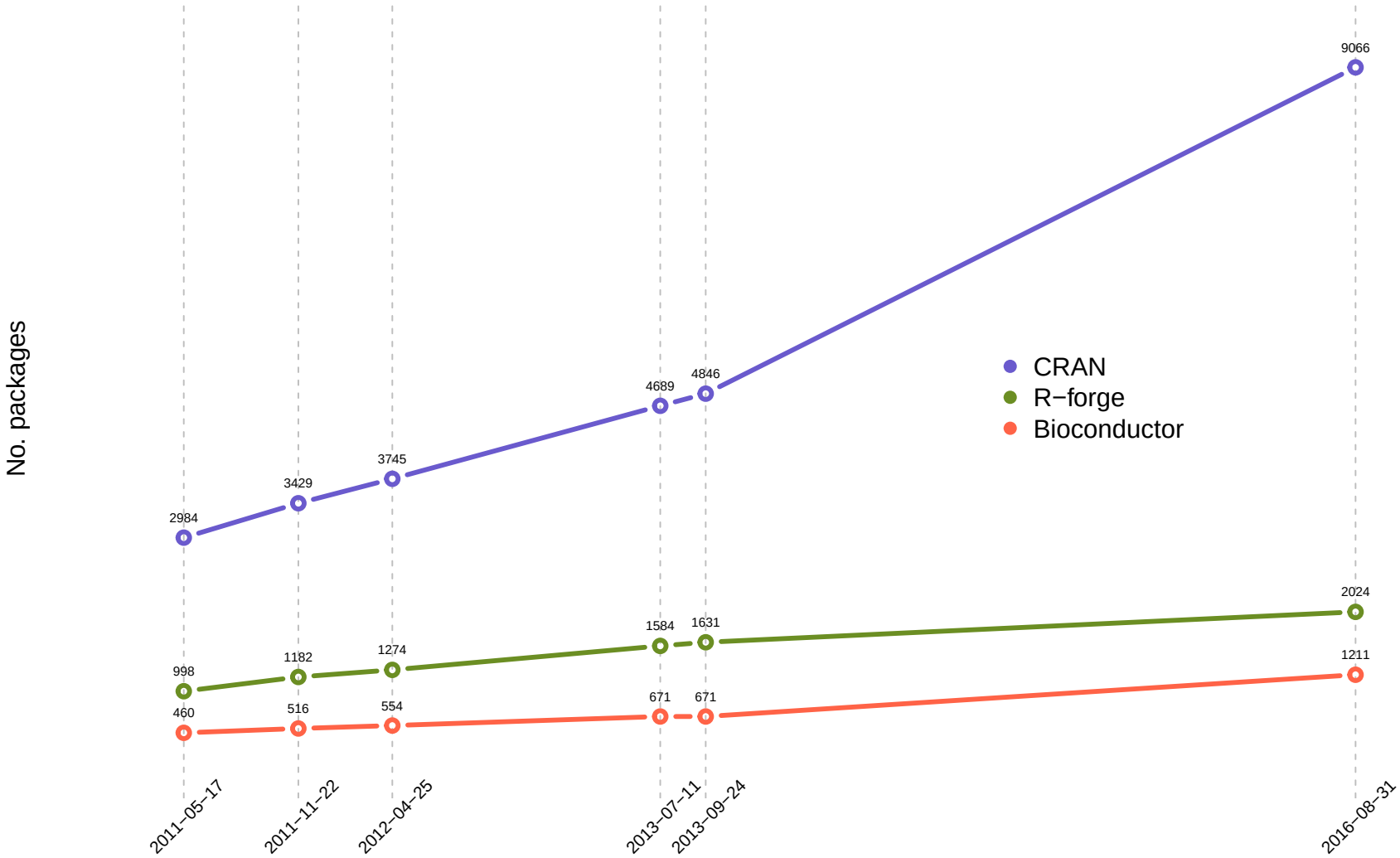
A brief history of R cted.

- open-source solution –> fast development,
- based on the S language created at the Bell Labs by **John Chambers** to to *turn ideas into software, quickly and faithfully*,
- inspired also by Lisp syntax (lexical scope),
- since 1997 developed by the R Development Core Team (~20 experts, with Chambers onboard),
- overviewed by The R Foundation for Statistical Computing
- learn more

The system of R packages – an overview

- developed by the community,
- cover several very diverse areas of science/life,
- uniformely structured and documented,
- organised in repositiries:
  - CRAN - The Comprehensive R Archive Network,
  - R-Forge,
  - Bioconductor,
  - GitHub.

R packages in the main repos



Advantages of using R

- a very powerful ecosystem of packages,
- uniform, clear and clean system of documentation and help,
- good interconnectivity with compiled languages like Java or C,
- free and open source, GNU GPL and GNU GPL 2.0,
- easy to generate high quality graphics.

Disadvantages of R

- a steep learning curve,
- sometimes slow,
- difficulties due to a limited object-oriented programming capabilities, e.g. an agent-based simulation is a challenge,
- cannot make a perfect espresso for you :-).

What a programming language is

A programming language is a formal computer language or constructed language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behavior of a machine or to express algorithms.[source: Wikipedia]

- We talk about the:
  - **syntax** – the form and
  - **semantics** – the meaning of a programming language.
- Languages can be of two major kinds:
  - **imperative** – a set of step-by-step instructions (R),
  - **declarative** – a clearly defined goal.

Programming paradigms

There many programming paradigms $\sim=$ styles of programming, e.g.:

- imperative:
  - literate (R, knitr, Sweavy, R Markdown),
  - procedural (R - functions),
  - ...
- declarative:
  - functional (R, λ-abstraction),
  - ...
- agent-oriented,
- structured:
  - object-oriented (R, S3 and S4 classes),
  - ...
- ...

Interpreted vs. compiled languages

- Computers understand the **machine code** not programming languages!
- Machine code is what the processor (CPU) understands.
- Thus, every computer language code has to be in some ways turned into the machine code.

Two major approaches exist to turn code in a particular language to the machine code:

- **Interpretation** – on-the-fly translation of your code, theoretically line-by-line. This is done every time you run your program and the job is done by a software called an **interpreter**.
- **Compilation** – your program is translated and saved as a machine code and as such can be directly executed on the machine. The job is performed by a **compiler**.

Elements of a programming language

Think of a program as a flow of data from one function to another that does something to the data.

- type system – definition of legal types of data,
- syntax – the form defined by the grammar,
- semantics – the meaning.

The syntax

- Syntax is the form, defined by **grammar** (typically Chomsky II == context-free grammar) like:
  - 2 * 1 + 1
  - (+ (* 2 1) 1)

Lisp is defined by the following grammar (BNF or Bakus-Naur Form):

```
expression ::= atom | list
atom       ::= number | symbol
number     ::= [+-]?['0'-'9']+
symbol     ::= ['A'-'Z''a'-'z'].*
list       ::= '(' expression* ')'
```

The semantics

Semantics is the meaning, a gramatically correct sentence does not necessarily have a proper meaning:

- *"Colorful yellow train sleeps on a crazy wave."* – has no generally accepted meaning.
- *"There is $500 on his empty bank acount."* – cannot evaluate to true.
- *Static semantics* – in compiled languages, e.g. checking that every identifier is declared before the first use or that the conditionals have distinct predicates.
- *Dynamic semantics* – how the chunks of code are executed. For instance lazy vs. eager evaluation.

The type system

- Typed vs. untyped languages.
  - 1 - integer
  - 1.0 - float
  - “1.0” - string
- Static vs. dynamic typing.
  - Static - type determined before execution, declared by the programmer (manifestly-typed) or checked by the compiler (type-inferrred) earlier:

```
integer i    # Declaration
i = 1        # Initialization
```

  - Dynamic - type determined when executing.
- Weak vs. strong types.
  - Weak - 1 can be either an int 1 or a string “1”
  - Strong - types cannot change.

Types – ERROR checking!

A more formal description of R

- Interpreted – it is every time translated by the interpreter.
- Dynamically typed – you do not declare types.
- Multi-paradigm:
  - array – works on multi-dimensional data structures, like vectors or matrices,
  - functional – treats computation as evaluation of math functions,
  - imperative – the programmer specifies how to solve the problem,
  - object-oriented – allows working with objects: data + things you can do to the data,
  - procedural – structure is organised in procedures and procedure calls, e.g. functions and
  - reflective – the code can modify itself in runtime.

So how to program?

Divide et impera – Divide and rule.

Top-down approach: define the big problem and split it into smaller ones. Assume you have solution to the small problems and continue – push the responsibility down. Wishful thinking!

```
You've got a csv file that contains data about people:
year of birth, favorite music genre and the name of
a pet if the person has one and salary. Your task is
to read the data and, for people born in particular
decades (..., 50-ties, 60-ties, ...), compute the
mean and the variance salary and find the most
frequent pet name.
```

Problem decomposition 1

This task can be decomposed into:

- read data from csv file,
- split the data into age classes based on the decade of birth,
- **compute the mean and the variance salary per class**,
- find the most frequent pet name per class.

Problem decomposition 2

To compute an the mean you have to: *sum all values*, *divide the sum by the number of values* – simple enough, we can program it right away.

To compute the variance you need to first refresh the formula:

$$Var(X) = \frac{1}{n}\Sigma_{i=1}^{n}(x_i - \bar{x})^2$$

Thus, you realise that you need to compute the mean, but you know how to do this from the previous point. So, instead of coding computation of the mean twice, make a function that you can reuse! Lazines is the major driving force of a programmer!

Let's put it down!

Pseudocode 1

$$Var(X) = \frac{1}{n}\Sigma_{i=1}^{n}(x_i - \bar{x})^2$$

Task: create the computeMean procedure that computes the mean for a sequence of numbers

Input: a sequence of numbers, e.g.: {1, 4, 5.7, 42357.533, 42}. Wait, isn't it a *vector*?

Output: the computed mean, a single number, that is what we want our procedure to return.

```
function computeMean(aVector) {
    sum = sum all numbers in aVector
    count = count how many numbers are in aVector
    theMean is: sum / count
    return theMean
}
```

Pseudocode 2

```
function computeMean(aVector) {
    sum = 0
    for every number n in aVector {
      sum = sum + n
    }
    count = count how many numbers are in aVector
    theMean is: sum / count
    return theMean
}
```

Pseudocode 3

But we realise that we can do counting in the same loop as addition:

```
function computeMean(aVector) {
    sum = 0
    count = 0
    for every number n in aVector {
      sum = sum + n
      count = count + 1
    }
    theMean is: sum / count
    return theMean
}
```

Pseudocode 3

Returning to our initial task of computing variance:

$$Var(X) = \frac{1}{n}\Sigma_{i=1}^{n}(x_i - \bar{x})^2$$

```
function computeVariance(aVector) {
  sumOfSq = 0
  count = 0
  for every number n in aVector {
    sumOfSq = sumOfSq + square((n - computeMean(aVector)))
    count = count + 1
  }
  variance = 1/count * sumOfSq
  return variance
}
```

Pseudocode 4

Please note, that we do not care about writing a function for addition, it is *fixed* for us by the '+' operator. Languages differ in what is already available in them as such. In R, we have a ready function for computing variance: `var(x)`.

Summary

So far, we have learnt about:

- what R is and what it is not,
- history of R,
- the system of packages,
- advantages and disadvantages of the language,
- definition of a programing language,
- elements of a programing language (types, syntax and semantics),
- programing paradigms,
- wishful thinking,
- problem decomposition,
- pseudocode.

Quite a bit, right?