Introduction to R Programming

SS 2016

Dr. Holger Fröhlich
Global Statistical Sciences
UCB BioSciences GmbH, Monheim

What is R?

- R is an integrated *environment* of software facilities for
  - data handling and storage
  - data manipulation
  - Numerical calculation, specifically with matrices
  - data analysis
  - graphical display
  - programming

Why R?

- Huge collection of tools
  - >1000 sofware packages to date
  - www.cran.r-project.org, www.bioconductor.org
  - Areas:
    - data manipulation, software development, plotting
    - Statistics, data mining, machine learning
    - Bioinformatics
      - Sequence analysis
      - -omics data analysis
  - Search engine rseek.org for finding R-packages

- Easy to handle graphical facilities
- Rapid prototyping
- Easy integration of C++ code for time critical calculations

- Recommended programming environment: RStudio

R command line system

- R can be used in two ways:
  - Interactive mode via a command line / shell (use arrows to recalling your command history)
  - Running of complete R scripts (to be written in an editor)
- R shell is active once you have started R (quit via `q()`)

The R help system

- R has an inbuilt help facility similar to the man facility of UNIX
- To get help for a particular command type (e.g. solve) type `?solve` in the R console
- An html based help system, which is displayed in your web browser is started via `help.start()`
- You can search the help system for a particular topic or command via `help.search`

The R language

- Case sensitive
- Allowed: all alphanumeric symbols, '.' and '_',
  - any variable name must start with '.' or a letter, and if it starts with '.' the second character must not be a digit.
- Commands are separated either by a semi-colon (';'), or by a newline.
- Elementary commands can be grouped together into one compound expression by braces ('{' and '}').
- Comments*:* '#'

Program Control: Conditional Execution

- Conditional execution: `if(A) expr1 else expr2`, where `A` is a logical expression using all kinds of allowed logical operators (see logical vectors)
  - Use grouping via braces {,} to group individual expressions in expr1 and expr2
  - Example:

```
if(x > 5){
   y <- x
   z <- x / 10
}
else{
   cat(„x is not larger than 5\n“)
}
```

Program Control: Loops

- Loops:
  - `for (name in expr_1) expr_2`, where `name` is a loop variable and `expr_1` a vector, e.g. 1:10

Example:

```
for(i in 1:10){
   j = i*i
   cat("i = ", i, "j = ", j, "\n")
}
```

  - `repeat expr`
  - `while (condition) expr`
  - The `break` statement can be used to terminate any loop, possibly abnormally. This is the only way to terminate `repeat` loops.

Writing own Functions

- A function is defined by an assignment of the form

name <- function(*arg_1*, *arg_2*, ...) *expression*

Example

```
mymean <- function(x){
   m <- 0
   for(i in 1:length(x))
      m <- m + x[i]
   m / length(x) # returns the function value
}
```

- Any ordinary assignments done within the function are local and temporary and are lost after exit from the function
- Hence: `m` is only known within function `mymean`

The R workspace

- Created and manipulated entities are known as *objects:*
  - Variables
  - arrays of numbers
  - character strings
  - Functions
  - more general data structures built from such components
- `ls()` displays all objects currently stored (*workspace*)
- Remove objects from workspace:

```
rm(x, y, z, ink, junk, temp, foo, bar)
```

- Storage on disk via `save` and later on reloadvia `load`.

Simple Data Structures

- *Vector:* array of numbers, strings or any other kinds of objects.

```
x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

- Numeric vectors can be used in arithmetic expressions
  - operations are performed element by element
  - Constants are implicitly repeated

Vector Arithmetics

- Elementary arithmetic operators: +, -, *, / and ^ for raising to a power
- All common arithmetic functions are available: `log`, `exp`, `sin`, `cos`, `tan`, `sqrt`, and so on, all have their usual meaning.
- Other common operations:
  - `max`, `min` select the largest and smallest elements of a vector respectively.
  - `range` is a function whose value is a vector of length two: `c(min(x), max(x))`.
  - `length(x)` is the number of elements in x
  - `sum(x)` gives the total of the elements in x
  - `prod(x)` their product.
  - `sort(x)` for sorting x
  - `mean(x)` the average and `var(x)` the variance defined as:

`sum((x-mean(x))^2)/(length(x)-1)`

Number Sequences, Logical Vectors

- `1:30` is the vector `c(1, 2, ..., 29, 30)`
  - Similarly: `30:1`
- Function `seq` allows for more flexible generation of sequences
- `rep` can be used to repeat numbers

- Vectors can also be logical

```
tmp <- x > 3
```

- temp is a vector of the same length as x with values FALSE corresponding to elements of x where the condition is *not* met and TRUE where it is
- Logical vectors can be combined using `&` (and), `|` (or), `!` (not), `==` (equal), `!=` (unequal), `<`, `>`,`<=` (less or equal), `>=` (greater or equal),

Character Vectors and Indices

- Vectors can be also composed of strings (type *character* in R)

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7",
  "Y8", "X9", "Y10")
```

- Several strings may be pasted together into one string via `paste`

- We can give names to vector elements via names

```
names(x) <- letters[1:10]
```

- Vectors can be indexed
  - `x[1:10]` returns sub-vector of the first 10 elements of x
  - `x[-c(1,5)]` excludes the first and the fifth element
  - Indexing via a logical vector of same length of x: `x[x > 5]` returns all elements of x being larger than 5
  - indexing via names, e.g. `x[c("a","c")]`

Matrices

- Matrices are two-dimensional arrays

```
M = matrix(0, ncol=5, nrow=2)
```

→ creates a 2 x 5 matrix of all 0s

- `dim(M)` shows the dimensions of the matrix (here 2 5)
- Columns and rows can be given names via `colnames` and `rownames`
- Indexing:
  - Index vectors, e.g. `M[1:3, 1:2]`, `M[c(1,3),1]`, ...
  - Index matrix, e.g. `M[M > 5]` returns a numeric vector containing all elements of M being larger than 5
  - Indexing via names, e.g. `M["a",c("B","C")]`
- Matrix algebra:
  - +,-,*,/,^ work on an *element-wise* basis
  - matrix product: `%*%`
  - `t(M)`: transpose matrix M (exchange columns and rows)

Matrices

- All kinds of linear algebra operations available, e.g. `eigen` (eigen vector decomposition), `qr` (QR decomposition), `chol` (Cholesky decomposition), `solve` (solve a linear equation system), ...
- Matrices can be concatenated via
  - `rbind(M, M2)`: extend rows of M
  - `cbind(M, M2)`: extend columns of M
  - Ensure that matrix dimensions of M and M2 fit!
- Matrices can be converted into vectors: `as.vector(M)`
  - Concatenates **columns** of M into one vector

Lists

- *list*: ordered collection of objects (*components)*
- Components can be of different type

```
Lst <- list(name="Fred", wife="Mary",
  no.children=3, child.ages=c(4,7,9))
```

- Components are always *numbered/index*
  - `Lst[[1]], Lst[[2]], Lst[[3]]`
- If `Lst[[4]]` is a vector subscripted array, then `Lst[[4]][1]` is its first entry.
- `length(Lst)` gives the number of (top level) components.
- Components of lists can be *named* ➔ access component via its name, e.g. `Lst$name`
- Lists can be concatenated via `c()`

Other Important Data Types

- *array*: multi-dimensional generalizations of vectors / matrices
- *factor*: data type to handle categorial variables
- *data.frame*: matrix-like structure, in which columns can be of different types (numeric, categorical, character).

- Functions are also objects
- Users can define their own data types via the `class` function

Some Comments on Programming Style

- It is highly important to write your code in a clean and organized manner!

- Give variables meaningful names
- Think what you want to achieve with your program and decompose it in small and easy to implement functions
- Group functions in a logical way together and save them in separate .R files
- Each `{` should follow a tab insert in the next line so that you see, which braces match

good

```
fib <- function(n){
    F <- double(n)
    F[1] <- F[2] <- 1
    ...
}
```

bad

```
fib <- function(n){
F <- double(n)
F[1] <- F[2] <- 1
...
}
```

Choosing an Editor

- Don‘t use Word to write R scripts! This will make your life much more difficult than necessary.
- A good programming editor should have the following features
  - Syntax highlighting
  - Automatic checking, whether braces {,} are matching
  - Ability to organize R scripts into different projects
  - Possibility to execute R scripts directly with one button

- Recommendations
  - RStudio
  - Eclipse: requires installation of StatET package, more difficult to set up than Rstudio
  - Emacs (only for unix)

Further Reading

- Besides the basic functionality covered here, R offers a lot more things
- **Read the manuals on http://www.r-project.org/**

- This will also make the things discussed here more clear!