R and Simulation

Computational Statistics

Lesson 1

G. Bertarelli[1]

[1]Dept. of Economics & Management -University of Pisa

November, 9th, 2018

The presentation at a Glance

Contacts

- Gaia Bertarelli, Postdoc Researcher in Statistics, University of Pisa
- e-mail: gaia.bertarelli@ec.unipi.it
- phone: (+39) 050 2216229
- webpage: https://people.unipi.it/gaia_bertarelli/

Syllabus

- R fundamentals and programming;
- LaTeX and RStudio;
- Pseudo-random numbers and variates generation;
- Monte Carlo methods for numerical integration;
- Special Topics - Outlier Robust Finite Population Estimation.

The course main follows (in the first part of the program)
R and Introductionn to Simulation by Federico Andreis, University of Stirling.
and
Robert, Christian P., George Casella, and George Casella. *Introducing monte carlo methods with R* Vol. 18. New York: Springer, 2010.

Introduction

- Simulationa are a way to model / imitate events in different application.
- Two main kinds of simulations:
  - Deterministic: when it is possible to control all the variables that contribute to the simulation context;
  - Stochastic: the experiment of interest takes place under the influence of unknown events attributable to chance.
- In this course we focus on Stochastic Simulations, which imply the use of probability theory.

- A history of simulation can be written from many perspectives: *uses* of simulation (analysis, training, re-search); *types* of simulation models (discrete-event, continuous, combined discrete-continuous); simulation *programming languages* or environments and *application domains* or communities of interest (communications, manufacturing, military, transportation,political-economic).
- In Statistics, simulations are employed as estimation methods (Monte Carlo Markov Chain, Resampling, E-M algrithm) as well as in research (e.g., to analyse the properties of an estimator having a structure that is too complex to be handled analytically).
- The development of simulation is connected to the advancement of technology and computer science (Goldsman et al., 2010; Rosen, 2013).

Pros and cons of simulations

- While analytical framework can provide an exact result, simulations yield an empirical result, i.e. a number which is valid for the particular simulated experiment only.
- Using simulations it is possible to provide efficient numerical solutions when:
  - the structure of the model at study is too complex to exploit the analytical (exact) approach
  - an exact analytical result is obtainable only under strong working hypotheses that are unlikely to hold true in reality
  - analytical results exist only as approximations or asymptotically, therefore, in turn, not exact themselves.
- Whenever available - or obtainable with acceptable effort and expenses - the exact analytical result is beyond doubt to be preferred to the simulative result.

Naylor et al. (1966) present 12 reasons why simulations can be an adequate tool to obtain results:

- simulations make it possible to experiment systems with complex inner structures
- through simulations, it is possible to study the effect of changes on the behaviour of a system
- a thorough observation of a system we wish to simulate can lead to a better understanding of the system itself
- simulations can be used as a pedagogical device to educate students and users about theoretical and statistical analysis and applied decision theory
- simulation of real systems seems a useful tool to stimulate interest in the audience

- the very same experience of structuring and organizing a computer-based simulation might be even more useful than the results themselves, because it often suggests interesting insights on the system at study
- simulation of complex systems helps to better understand what the most important variables and their interactions are
- simulations allow to experiment new scenarios we have little knowledge of
- simulations might be useful as preventive tests for policies and decisions before risking real application
- simulations can give useful indications on how to split a complex system in less involved sub-systems

- simulations allow to study dynamic systems in time
- when new components are introduced a system, simulation can be useful to predict bottlenecks or other problems that might occur in the real system.

One of the most important characteristic of a computer-based simulation is that it is reproducible, i.e. it can be replicated under the same conditions as many times as wished. This is of particular relevance in that it allows to verify the effects of any variation artificially introduced in the experiment. Moreover, simulations are flexible, as opposed to the (usually strict) requirements and working hypotheses of the analytical framework

What about the cons?

- ★ simulative solutions are fundamentally imprecise and not exact
- ★ under the practical aspect, simulations are a slow and costly method to analyse a system
- ★ huge implementation effort and usually call for a considerable amount of machine-time to produce a result
- ★ simulations yield numerical solution, i.e. strictly connected to the specific simulated case, rather than general theoretical that are valid in general; in order to generalise the simulative results to wider classes of problems similar to the one at study, the only possibility seems to be to conduct series of simulations, manipulating the involved variables to recreate several characterizations of the real system under as many scenarios as possible, which leads to lengthier, costlier, and more complex simulations.

Why R?

- R is open-source *(free)*
- R is object-oriented, which allows for efficient handling of complex structures *(flexible and object oriented)*
- R provides a powerful interface to integrate programs written in other languages *(easily interfaceable)*
- R has a huge, active and constantly evolving users-based community *(community-supported)*
- R has amazing graphical capabilities *(highly graphically skilled)*.

R, with its ever-growing number of interfacing packages, provides a great help in making integration easier, even without specific knowledge of advanced computer science.

References about R

- official manuals here: `http://cran.r-project.org/`
- books: Adler (2010); Crawley (2012); Grolemund (2014); Matloff (2011)
- the whole *Use R!* series published by Springer, in which you can find specialized texts on variety of topics (we will use *Introducing Monte Carlo Methods with R* Authors: Robert, Casella.)
- the funny guide *The R Inferno*, by Patrick Burns, available here: `http://www.burns-stat.com/pages/Tutor/R_inferno.pdf`.
  '*If you are using R and you think you're in hell, this is a map for you*'.

The RStudio IDE

- When programming, the need to resort to other tools arises: for example, external text editors, file managers or web browsers.
- IDE (Integrated Development Environment)
- RStudio (`https://www.rstudio.com/`)
- In just one screen:
  1. a powerful text editor with syntax highlighting, brackets matching (VERY useful) and buttons/keyboard combinations to submit code snippets to the console directly (i.e., no need for copy-and-paste nonsense)
  2. a workspace browser, a data viewer and the commands history
  3. a file manager, a package manager, a plot tab (where you can browse all the plot you created, back and forth) and an integrated help tab.

Organize your work

- ★ There is no general rule on how to organize your working directories
- ★ I create a single directory for each project `ProjectName` and create then the following sub-directories
  - `ProjectName/Code`: this is were to save all the $.R$ code snippets
  - `ProjectName/Data`: $.txt, .xls, .csv, .dat, .spss, \ldots,$ raw and formatted data files in general go here
  - `ProjectName/Literature`: did you find a paper related to your project? Put it here!
  - `ProjectName/Out`: any output of your work, e.g. plots, text summaries, ...
  - `ProjectName/Varia`: anything else that doesn't belong to the other directories (project documentation, call for papers, to-do lists, ...).

Help functions

R provides a wealth of functions, datasets and packages (more than 6000 now), definitely too many to be able to keep trace of every functionality, syntax or purpose. Some commands that might come in handy are:

- `help()`: need to find out what function `xxx` does? simply enter `help(xxx)` in the command console (`?xxx` is an equivalent way to do so). Were you interested in the help for operators involving special symbols in their syntax (such as the matrix multiplication operator `%*%`), do not forget to put it between apostrophes: `?'%*%'`
- `help.search()`: equivalently, `??`, allows for searching the help system for documentation matching a given character string. Say we want to find every function in every loaded package that deals with optimization, then we should enter `??optimization`; this command also accepts regular expressions

- `apropos()`: accepts a string as an argument, and returns a character vector giving the names of all objects in the search list matching it; it might be useful to find, for example, all the functions containing the word glm (`apropos('glm')`)
- `data()`: looking for some data? just enter `data()` on the command line and a list of all the datasets availabe in the default package `datasets` and in the currently loaded libraries, together with a short description thereof, will appear. These datasets can then be readily examined with the help command (`?nameofthedataset`), or loaded into the working space (`data(nameofthedataset)`).

If you are having coding problems

- http://stackoverflow.com
- http://www.r-project.org/mail.html.

Packages handling

- R basic installation comes with a number of packages that have been deemed essential.
- However, you will often need to resort to specific functions and routines, that can be downloaded by one of the many repositories scattered around the world (wide web).
  - `install.packages()`: as the name states, it allows to install new packages, provided the exact name is matched, as is `install.packages('nameofthepackage')`
  - `library()`: if the package you need is available on your computer, you have to load it in the current R session, in order for its contents to be at your disposal; entering `library(nameofthepackage)` will do exactly this, making all the functions and datasets in the package available for usage. The `require()` command accomplishes the same task, but is intended specifically for use inside functions.

Objects and indexing

- R is an object-oriented programming (OOP) language
- An R object is basically a category, examples are: scalar, vector, matrix, array, list, time series, data frames, functions and graphics
- An object, say `x`, is characterized by a `mode(x)` that describes its contents and a `class(x)` that describes its nature
- An object's `mode` might be one of the following:
  - `null`: empty object
  - `logical`: `TRUE` or `FALSE`
  - `numeric`: a real number, such as $\pi, 2$ or $\sqrt{5} - 1$
  - `complex`: a complex number, such as $2 + 5i$
  - `character`: a string, such as `'normal'`, `'green'`, `'M'` or `'y=a+bx'`

R code

```
> x <- 10
> x
[1] 10
> mode(x)
[1] "numeric"
> x <- 10>5
> x
[1] TRUE
> mode(x)
[1] "logical"
```

Many functions exist, that can create objects of a specific kind, for example:

- `c()` defines a row vector of given elements
- `numeric()` defines an empty vector of given length
- `matrix()` defines a matrix of given number of rows and columns, and given elements
- `array()` defines an array of given dimensions and elements
- `data.frame()` defines a data-frame object
- `list()` defines a list object, i.e. a special array that can include elements with various modes

The function `str(x)` will compactly display the internal **str**ucture of an R object, and is a very useful command, especially when working with complex ones.

R code

```
> y <- list(a=c(1,2,3),b=matrix(c(1,2,3,4,5,6),3,2))
> y
$a
[1] 1 2 3

$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> mode(y)
[1] "list"
> str(y)
List of 2
 $ a: num [1:3] 1 2 3
 $ b: num [1:3, 1:2] 1 2 3 4 5 6
```

R code

```
> y$a            #extracts component 'a' from list 'y'
[1] 1 2 3
> y$b            #extracts component 'b' from list 'y'
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> dim(y$b)     #what are the dimensions of matrix 'b'?
[1] 3 2            #3 rows and 2 columns
> y$b[1, ]       #extracts the first row of matrix 'b'
[1] 1 4
> y$b[ ,2]       #extracts the second column of matrix 'b'
[1] 4 5 6
> y$b[1:2, ]#extracts rows from 1 to (:) 2 of matrix 'b'
     [,1] [,2]
[1,]    1    4
[2,]    2    5
> y$a[3] #extracts the third element of vector 'a'
[1] 3
```

It is possible to assign values to specific elements of an object:

R code

```
> y$a[3] <- 10 #assign to the third element of vector 'a'
> y$a          #the value 10
[1] 1 2 10
> y$b[,1] <- c(2,3,7) #replace the first column of matrix 'b'
> y$b                 #with a vector having components (2, 3, 7)
     [,1] [,2]
[1,]    2    4
[2,]    3    5
[3,]    7    6
```

An array is a matrix in dimensions higher than 2 (a *tensor*). Indexing works in the same way, allowing to extract and manipulate information at every level by means of the square brackets operators [,,. . . ,].

R code

```
> z <- array(runif(24),dim=c(4,3,2))
> #creates an array composed by two 4x3 matrices
> #filled by realizations from a U(0,1) r.v.
> z
, , 1
           [,1]       [,2]      [,3]
[1,] 0.50948231 0.09637384 0.3511716
[2,] 0.76222931 0.49445906 0.9239261
[3,] 0.07748602 0.24500671 0.9360242
[4,] 0.14746812 0.63679512 0.1754347

, , 2
          [,1]      [,2]      [,3]
[1,] 0.4685381 0.9641261 0.9762254
[2,] 0.3219655 0.5849373 0.7358713
[3,] 0.5257197 0.8853259 0.5509927
[4,] 0.7837449 0.4370338 0.9350366
```

R code

```
> z[,,1] #extracts the first matrix
           [,1]       [,2]      [,3]
[1,] 0.50948231 0.09637384 0.3511716
[2,] 0.76222931 0.49445906 0.9239261
[3,] 0.07748602 0.24500671 0.9360242
[4,] 0.14746812 0.63679512 0.1754347
> z[1,,2] #extracts the first row of the second matrix
[1] 0.4685381 0.9641261 0.9762254
> z[,2,1] #extracts the second column of the first matrix
[1] 0.09637384 0.49445906 0.24500671 0.63679512
> z[,1,1] <- 1:4
> #assign the values from 1 to 4 to the first column of
> #the first matrix
> z[,,1]              #print the (modified) first matrix
```

R math operators usually work element-wise

R code

```
> x <- c(1,2,5,8)              #defines a vector
> 2*x                      #multiplies every element of x by 2
[1]  2  4 10 16
> x^2                       #squares every element of x
[1]  1  4 25 64
> sqrt(x)                       #extracts square roots
[1] 1.000000 1.414214 2.236068 2.828427
```

Indexing allows to apply specific operations on chosen parts of an object

R code

```
> x <- matrix(1:25,5,5)
> .5*x[,1]       #multiplies the first column by 0.5
[1] 0.5 1.0 1.5 2.0 2.5
> x[1,]^(1/3) #extracts cube roots from the 1st row
[1] 1.000000 1.259921 1.442250 1.587401 1.709976
> x[,1]*x[,2] #element-wise multiplies first and 2nd column
[1]  6 14 24 36 50
```

Matrix Algebra

Special operators and functions exist for most of the matrix algebra. Let A and B be matrices, b a vector; then:

- `A*B`: element-wise multiplication
- `t(A)`: transpose of a matrix (A') or a vector (b')
- `A%*%B`: matrix multiplication (row by column)
- `A%o%B`: outer product (AB')
- `crossprod(A,B)`: matrix crossproduct, equivalent to `t(A)%*%B`
- `diag(A)`: returns a vector containing the elements of the principal diagonal; it can also be used to create diagonal matrices, refer to `?diag`
- `solve(A)`: inverse (A^{-1}) of a square matrix A
- `solve(A,b)`: returns the solution of the system $b = Ax$, i.e. $x = A^{-1}b$
- `ginv(A)`: Moore-Penrose Generalized Inverse of A (requires to load the `MASS` package)
- `eigen(A)`: returns eigenvalues and eigenvectors of A

- `svd(A)`: returns the Singular Value Decomposition of A
- `chol(A)`: returns the Choleski factorization of A
- `qr(A)`: returns the QR decomposition of A
- `cbind()`: combines matrices (vectors) horizontally, returns a matrix
- `rbind()`: combines matrices (vectors) vertically, returns a matrix
- `rowMeans(A)`: returns a vector of row means
- `rowSums(A)`: returns a vector of row sums
- `colMeans(A)`: returns a vector of column means
- `colSums(A)`: returns a vector of column sums

Working space, directories and I/O

★ All the objects you create are saved in a working space that can be browsed, modified and saved for future use.

- `ls()`: plain execution of this command returns the list of the objects in the working space
- `rm()`: this command removes an object matching its argument from the working space; say we wish to delete the object `xxx`, then we would enter `rm(xxx)` in the console. Particularly useful is the combination `rm(list=ls())`, that will remove all the objects from the working space
- `getwd()`: execution of this command returns the current working directory. Should the need to use a different working directory arise, the `setwd()` command would change the default for the current session. Setting one's own working directory as default is particularly useful in that every input/output (I/O) procedure will then refer to it, thus making destination specifications shorter and more convenient.
- `list.files()`: returns a character vector containing the names of the files in the current working directory
- `save.image()`: if you want to save the current workspace to a file with extension `.Rdata`.
- `objects.size()`: returns the (estimated) amount of memory that is being used to store an `R` object.

- Many functions exist, that can read external files into the R working space, such as raw data (usually text files):
  - `scan()`, `read.table()` and `read.csv()`: check their help file for syntax and usage.
- It is worth noting that many packages exist, among them we mention the `foreign` and the `readr` packages, that contain specific routines to import particular data formats such as, for example, those native of Excel, Minitab, S, SAS, SPSS and Stata.
- For what concerns text output, R provides many ways to create `.txt` and `.csv` files:
  - `write()`, `write.table()`, `write.csv()` and `sink()`: check their help file for insights.

- Output can also take the form of screen text and plots:
  - `print()`: prints its argument on screen
  - `cat()`: prints its argument by con**cat**enation, it is useful for producing output in user-defined functions (for example, printing to screen at what step of a loop your routine is, or printing in a nice way your function results)
  - `paste()`: often used as an argument to other functions, it allows to paste together, as a character string, multiple objects (useful, for example, for plot labels)
  - `plot()`: the base R function for plotting objects, see help
- `jpg()`, `pdf()`, `png()`, and `postscript()`; read the related help files and remember to end every such operation with the `dev.off()` command.

- Other graphical tools include: `points()`, `lines()`, `segments()` and `abline()`, that allow you to add, respectively, points, curves, segments and straight lines to your plots
- `curve()` command will allow you to plot functions of one variable over a chosen region of the xy plane
- http://rgraphgallery.blogspot.it/

Control structures, loops and miscellanea

- R is in general slower than other languages, therefore a proper use of flow operators is of great importance
  - `!`, `&&`, `||`, `...`: logical operators allows you to evaluate the truth (logical) value of an R object, check the help for the most common ones by entering `?'!'`.
  - `for`: for loops can be used to loop through the values of an object. The command is of the form `for (<index> in <vector>) {<statements>}`. The expressions between curly brackets are executed seperately for each value in the vector

R code

```
> x <- c('a','b','c')
> for (i in x) {
+   print(paste("Index=",i))
+ }
[1] "Index=a"
[1] "Index=b"
[1] "Index=c"
```

- For loops are very slow. You can improve code readability over more efficient solutions such as the *apply functions
- if/else: sometimes, a statement should be executed only if a certain condition is met. In this case, the if structure can be used, and complemented by the else control. It is built-up as follows: if (<logical expression>) {<statements1>} else {<statements2>}; if and else can be nested

R code

```
> x <- 10
> if(x > 9) {
+   print("x␣is␣larger␣than␣9")
+ } else if(x > 7) {
+   print("x␣is␣larger␣than␣7,␣but␣not␣larger␣than␣9")
+ } else {
+   print("x␣is␣not␣larger␣than␣7")
+ }
[1] "x␣is␣larger␣than␣9"
```

★ `ifelse()`: If you need to set, for example, values of variables based on a vector of logical conditions, then the `ifelse` statement (which basically is a vectorized version of `if`) is the way to go. The syntax is:
`ifelse(<condition>,{<yes statements>},{<no statements>})`

R code

```
> x <- -2:2
> ifelse(x>=0, x, -x)
[1] 2 1 0 1 2
```

★ This is equivalent to calling abs(x), i.e. the absolute value function.

★ while: it is one of the simplest looping structures, whose syntax is `while(<condition>) {<statements>}`. It is used to repeat `<statements>` while `<condition>` remains `TRUE`. For example, consider the following loop that stops as soon as a random draw from a Bernoulli distribution with parameter $p = 0.5$ equals 1 and stores the number of draws needed to reach that point

R code

```
> p <- 0.5 # probability of drawing a 1
> b <- 0        # result of the draw
> number <- 0   # number of draws
>
> while(b != 1)
+ {
+   b <- rbinom(1, 1, p)    # draws a Bernoulli variate
+   number <- number + 1    # increments the counter
+ }
>
> print(number)  #prints to screen the number of draws
```

- repeat: this structure repeats the commands in its body until a break statement is reached

R code

```
> k <- 0
> repeat {
+    k <- k + 1
+    if(k > 3) break
+    cat(k, "\n")
+ }
1
2
3
>
```

- tryCatch(): it provides a mechanism for handling with errors without breaking the loop
- system.time(): it takes an expression as argument and returns the CPU times spent to run it
- source(): You can save snippets of code in files other than your current one, and then source them for execution in the current session.

Statistical functions

- The R language has been originally written by statisticians
- R has implemented all sort of statistical routines to perform from the simplest to the most complex task

Basic statistical functions

- summary(): a generic function that, depending on the class of its argument outputs a summary thereof.

R code

```
> x <- iris[,1]
> y <- iris[,2]
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.300   5.100   5.800   5.843   6.400   7.900
```

R code

```
> mod <- lm(y~x)
> summary(mod)
Call:
lm(formula = y ~ x)
Residuals:
    Min       1Q   Median       3Q      Max
-1.1095 -0.2454 -0.0167  0.2763  1.3338
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.41895    0.25356   13.48   <2e-16
x           -0.06188    0.04297   -1.44    0.152
---
Residual standard error: 0.4343 on 148 degrees of freedom
Multiple R-squared:  0.01382,Adjusted R-squared:0.007159
F-statistic: 2.074 on 1 and 148 DF,  p-value: 0.1519
```

- `mean()`: generic function for the (possibly trimmed) arithmetic mean. `colMeans()` and `rowMeans()` functions for use with matrices.
- `var()`, `cov()`, `cor()`: these functions compute the variance, covariance and correlation of, say, `x` and `y` if these are vectors; `sd()` will compute the standard deviation
- if the argument to `cov()` (`cor()`) is a `matrix` or a `data.frame` object, the variance/covariance (correlation) matrix is returned

Probability distributions

- ⋆ The default R release features many classic probability distribution functions, enter `?distributions` to see them.
- ⋆ By convention, all functions related to a particular distribution, say `DISTR`, are encoded as follows:
  - `dDISTR()`: density function (with respect to an appropriate probability space)
  - `pDISTR()`: distribution function
  - `qDISTR()`: quantile function
  - `rDISTR()`: random variates generation

Example: the Central Beta random variable

- ★ The related functions are then `dbeta()`,`pbeta()`,`qbeta()`, and `rbeta()`
  - `dbeta(x, shape1, shape2)`: returns the value in x of the density function of a Beta random variables with parameters (shape1,shape2).

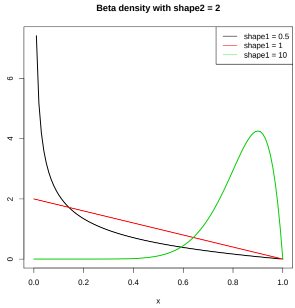
R code

```
> #value of the Beta(2,2) density in x=0.5
> dbeta(0.5, 2, 2)
[1] 1.5
```

- Plot the Beta density for different combinations of its parameters:

R code

```
> curve(dbeta(x,.5,2),ylab='',
+ main='Beta␣density␣with␣shape2␣=␣2',lwd=2)
> for (i in c(1,10)) curve(dbeta(x,i,2),add=T,
+ col=round(i+1),lwd=2)
> legend('topright',legend=c('shape1␣=␣0.5',
+ 'shape1␣=␣1','shape1␣=␣10'),col=c(1,2,11),lty=1)
```



- pbeta(q, shape1, shape2): returns the cumulative density function of a Beta distribution with parameters (shape1,shape2) in correspondence of quantile q

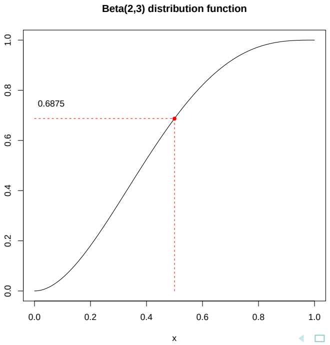
R code

```
> beta.value <- pbeta(0.5,2,3)
> beta.value
[1] 0.6875
```

- It is possible, for example, to visualize the quantile on the distribution function plot:

R code

```
> curve(pbeta(x,2,3),ylab='',
+         main='Beta(2,3)␣distribution␣function')
> points(0.5,beta.value,pch=16,col='red')
> segments(c(0,.5),c(beta.value,0),c(.5,.5),
+         c(beta.value,beta.value),lty=2,col='red')
> text(.06,beta.value+.06,beta.value)
```



- qbeta(p, shape1, shape2): returns the quantile of order p of a Beta distribution with parameters (shape1,shape2).

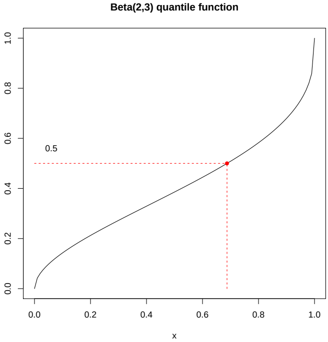
R code

```
> qbeta(beta.value,2,3)
[1] 0.5
```

- If we want to visualize the quantile function

R code

```
> curve(qbeta(x,2,3),ylab='', main='Beta(2,3)␣quantile␣function')
> points(beta.value,0.5,pch=16,col='red')
> segments(0,0.5,beta.value,.5,beta.value,lty=2,col='red')
> text(.06,0.5+.06,0.5)
```



- rbeta(n, shape1, shape2): generates n realizations from a Beta distribution with parameters (shape1,shape2).
- How this is accomplished will be described in the next lesson. Now, imagine we wish to draw a certain number of random variates from this law and compare their distribution with the theoretical density of a Beta. First, we obtain the draws

R code

```
> x <- rbeta(1000,2,3)#draws 1000 variates from Beta(2,3)
```

- ⋆ Check whether the sample mean and variance are close to the theoretical values. Recall that, if $X \sim \text{Beta}(\alpha,\beta)$, then $E(X)=\frac{\alpha}{\alpha+\beta}, V(X)=\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$;
- ⋆ in this case, since $X \sim \text{Beta}(2,3)$ we would then expect $E(X)=0.4, V(X)\approx 0.04$

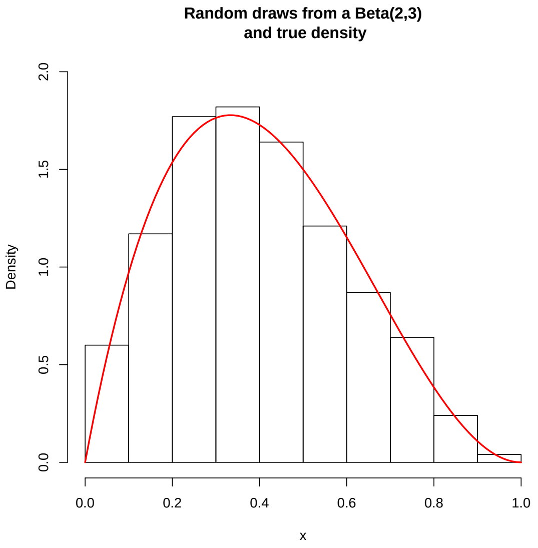
R code

```
> mean(x)
[1] 0.4085936
> var(x)
[1] 0.04034115
```

- Plot the distribution of the random variates and compare it with the true density:

R code

```
> #plots the histogram
> hist(x,freq=F,xlim=c(0,1),ylim=c(0,2),
+       main='Random␣draws␣from␣a␣Beta(2,3)\n␣and␣true␣density')
> #overimposes the theoretical density
> curve(dbeta(x,2,3),
+       col='red',add=T,lwd=2)
```


Random draws from a Beta(2,3)
and true density
Density
0.0
0.5
1.0
1.5
2.0
0.0
0.2
0.4
0.6
0.8
1.0
x

Statistical modeling

- Several functions exist, that can handle the most common statistical models;e.g.:
  - `lm()`: fits classic linear models, and provides a wealth of statistics on residuals, as well as hypothesis testing on the model's parameters
  - `glm()`: fits generalised linear models, and takes as input the description of the linear predictor and of the error term; again, the most common model-related statistics are computed as a by-product of the routine
  - `anova()`: based on one or more previously fitted model objects, computes the related analysis of variance (or deviance) tables.
- There are also routines for survival analysis, longitudinal models, time series, multilevel modeling, partial least squares, structural equations,. . . !

Hypothesis testing

- The base R release contains functions for many of the most common statistical tests, such as, for example, `t.test()` (for one- and two-sample t-test), `ks.test()` (for the Kolmogorov-Smirnov one- and two-sample tests), and `wilcox.test()` (for the Wilcoxon-Mann-Whitney tests)
- Some testing routing will be used during Lessons 2 and 3.

Statistical plots

- `plot()`: produces graphical representations based on the `class` of its argument
- `hist()`: it plots an histogram based on a vector of observations
- `boxplot()`: produces box-and-whisker plot(s) of the given (grouped) values, based on a `data.frame` or `list` object
- `density()`: computes kernel density estimates based on a vector of observations
- `acf()`: computes and plots estimates of the autocovariance or autocorrelation function based on a vector of observations. Useful when working with time series, stochastic processes, and when evaluating a random number generator
- `pairs()`: plots the pairwise scatterplots based on a `matrix` or `data.frame` object, useful for a first visual inspection of a dataset, or to summarize dependence between pairs of variables
- `image()`: creates a grid of rectangles with colors corresponding to the values in a given `matrix` object (3-dimensional data)
- `contour()`: creates a contour plot, or add contour lines to an existing plot

Writing R functions

- One of the most appealing features of R, is that everybody can create new functions and packages to be made available to all other R users;
- Most of R is actually written in R (apart from a few routines that are written in C for efficiency reasons), which means that you could basically take every function and suitably modify it to meet your needs
- An R function is defined by an assignment such as:

R code

```
MyFunction <- function(arg1, arg2, ...) {
        expression
        ...
        expression
        value
}
```

- `expression` is an R statement (possibly involving other functions) that uses some of the arguments `arg1, arg2, ...` to compute a `value`, that is the outcome of the function
- The curly braces indicate the beginning and the end of the function; it is also possible to set default values for the arguments
- a function should always output a `value`, since everything that happens within those curly braces (unless otherwise specified) is local to the function itself and, as such, temporary

Example 1

★ Create a function that:

1. takes two vectors as arguments
2. checks if they are of the same length, otherwise returns an error message

R code

```
length.check <- function(x,y) {#x and y are arg1 and arg2
  if (length(x)==length(y)){#the logical condition to be met
    print('Same␣length.')#value condition is TRUE
  } else {
    print('Different␣length.')#value if condition is FALSE
  }
}
```

★ Test our function

R code

```
> a <- 1:10
> b <- letters[2:11]
> c <- runif(15)
> length.check(a,b)
[1] "Same␣length."
> length.check(a,c)
[1] "Different␣length."
> length.check(b,c)
[1] "Different␣length."
```

Example 2

★ We now wish to write a function that:

1. has two arguments, of which the second one defaults to the value 1
2. computes the $k-$th sample moment of the first object (suppose it's a vector of sample observations), where k is defined by the second argument
3. outputs the computed value

R code

```
#By setting y=1 in the arguments,
#we define its default value
k.moment <- function(x, y=1) {
        value <- mean(x^y)       #stores the result

        value              #outputs the result

}
```

★ Test our function

R code

```
> a <- 1:10
> k.moment(a,1)
[1] 5.5
> k.moment(a)
[1] 5.5
> k.moment(a,2)
[1] 38.5
> k.moment(a,3)
[1] 302.5
```

Example 3: probability law not already implemented in R

- ⋆ Reciprocal Distribution
  - continuous random variable with bounded support $[a, b]$ $(0 < a < b)$ and density $f(x; a, b) = [x \ln(b/a)]^{-1}$
- ⋆ Code for:
  - density function
  - distribution function: it is easily obtained by integration and equals $F(q; a, b) = \ln(q/a)[\ln(b/a)]^{-1}$
  - quantile function: it is $Q = F^{-1}$

R code

```
#density function
drec <- function(x,a,b) {
 #checks constraints
  stopifnot(x>=a&&x<=b&&0<a&&a<b)
 #outputs density value
  (x*log(b/a))^(-1)
}
```

R code

```
#distribution function

prec <- function(q,a,b) {

 #checks constraints

  stopifnot(0<a&&a<b)

 #if q<=a, returns 0, otherwise
 #if q>=b, returns 1, otherwise
 #returns cdf value

  ifelse(q<=a,0,
         ifelse(q>=b,1,
                log(q/a)/log(b/a)))

}
```

R code

```
#quantile function
qrec <- function(p,a,b) {

  #checks constraints

  stopifnot(p>=0&&p<=1&&0<a&&a<b)

  #computes q numerically by
  #finding the zero of 'F(q)-p'
  #in [a,b], and returns it

  uniroot(
        function(x) prec(x,a,b)-p,
                lower=a,upper=b)$root

}
```

R code

```
> a <- c(1,2.5,5)
> drec(a,1,10)
[1] 0.4342945 0.1737178 0.0868589
```

R code

```
> a <- seq(0,1,.3)
> qrec(a,10,15)
[1] 10
Warning messages:
1: In if (is.na(f.lower)) stop("f.lower␣=␣f(lower)␣is␣NA") :
the condition has length > 1
and only the first element will be used
2: In if (is.na(f.upper)) stop("f.upper␣=␣f(upper)␣is␣NA") :
the condition has length > 1
and only the first element will be used
3: In if (f.lower * f.upper > 0)
stop("f()␣values␣at␣end␣points␣not␣of␣opposite␣sign") :
the condition has length > 1
and only the first element will be used
```

- qrec() internally calls uniroot(), that does not accept vector-valued functions as argument
- to solve this, it suffices to Vectorize the function

R code

```
> qrec <- Vectorize(qrec)
> qrec(a,10,15)
[1] 10.00000 11.29347 12.75426 14.40397
```

Exercises

- Using the `seq()` function, construct the following sequences:
  1. integer values from 1 to 10
  2. 32 equi-spaced values from 1 to 10
  3. values distant 0.15 from eachother, ranging from 1 to 10; how many of them are there? (Tip: use `length()`).
- Using the `replicate()` function, obtain a vector of 1000 realizations of means from uniform samples of size 100 each. Plot the histogram of the standardized observation and superimpose a Standard Normal curve (tip: set `freq=FALSE` when plotting the histogram).
- Read the help for the `iris` dataset and describe the variables it contains. Create a two-pages .pdf document file with:
  1. the pairwise scatter plots for all the involved variables, with different colours indicating different species on the first page
  2. the plot of the kernel density estimate of `Petal.Length`, regardless of the species and the histogram of `Sepal.Length` for the `versicolor` species only on the second page (tip: `?density`).

★ Read the help for the `iris` dataset and describe the variables it contains. Create a two-pages .pdf document file with:

1. the pairwise scatter plots for all the involved variables, with different colours indicating different species on the first page
2. the plot of the kernel density estimate of `Petal.Length`, regardless of the species and the histogram of `Sepal.Length` for the `versicolor` species only on the second page (tip: `?density`).

★ Write a function that:

1. accepts two arguments `a` and `b`, that must be non-negative integer with default value 1
2. plots the function $f(x; a, b) = |cos(e^{-ax^2+bx})|$ on the range $x \in [-1, 1]$
3. returns the value of $\int_{-1}^{1} f(x; a, b)dx$ on screen and saves it in a .txt file in the current working directory.

References I

Adler, J. (2010). *R in a nutshell: A desktop quick reference*. " O'Reilly Media, Inc.".

Crawley, M. J. (2012). *The R book*. John Wiley & Sons.

Goldsman, D., R. E. Nance, and J. R. Wilson (2010). A brief history of simulation revisited. In *Proceedings of the winter simulation conference*, pp. 567–574. Winter Simulation Conference.

Grolemund, G. (2014). *Hands-On Programming with R: Write Your Own Functions and Simulations*. " O'Reilly Media, Inc.".

Matloff, N. (2011). *The art of R programming: A tour of statistical software design*. No Starch Press.

Naylor, T. H., J. L. Balintfy, D. S. Burdick, and K. Chu (1966). Computer simulation techniques. Technical report, Wiley New York.

Rosen, K. (2013). The history of simulation. In *The comprehensive textbook of healthcare simulation*, pp. 5–49. Springer.