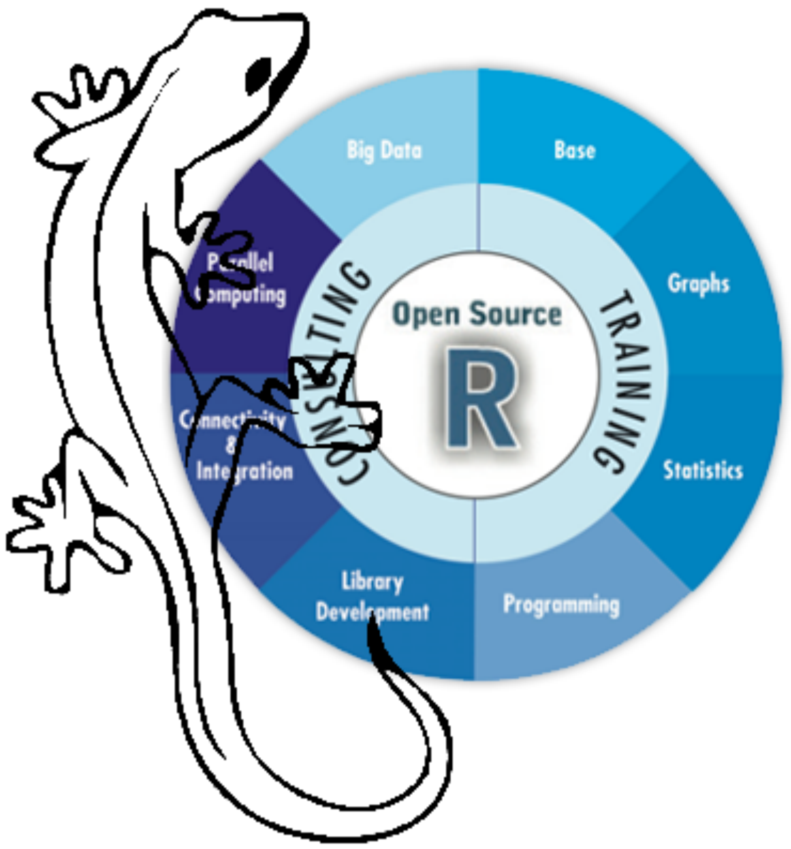
Andrea Spanò

RAMARRO

R for Developers



Contents

Acknowledgment

I have taken efforts in writing this manual. However, it would not have been possible without the sharing of knowledge, information, ideas, doubts and even criticisms of many individuals on the internet. I would like to extend my sincere thanks to all of them.

I would like to thank my colleagues: *Nicola Sturaro* and *Enrico Pegoraro* for their contributions and for the long hours they spent checking and reviewing my messy writing.

I am also grateful to: *Bill Venables* and *John Chambers* for their publications on R that provided solid foundations to my knowledge on this subject. For who concern environments, I found http://obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/ a good reference in this topic.

I would also like to express my special gratitude to Hadley Wickham for providing and sharing his research. Without his contribution (http://adv-r.had.co.nz) most parts of this manual would never have been written.

I finally express my sincere excuses to all researchers and R enthusiasts I have *borrowed* any knowledge from without mentioning them. This was not intentional, simply I had not always tracked my sources. If this is the case, please contact me directly and I will be more than happy to include any appropriate reference in this manual.

Chapter 1

Introduction

What is `R`? Possibly a language and environment for statistical computing and graphics but eventually R is much more. R is the result of the concerted efforts of many scientists from a wide variety of fields all of them truly convinced that the open source model could be a *`better`* way for deploying ideas and knowledge to a large community.

The `R` Foundation is a not for profit organization working in the public interest. It has been founded by the members of the R Development Core Team in order to provide support for the R project and other innovations in statistical computing.

The `R` Foundation believes that `R` has become a mature and valuable tool and they would like to ensure its continued development and the development of future innovations in software for statistical and computational research.

Provide a reference point for individuals, institutions or commercial enterprises that want to support or interact with the R development community.

The R Foundation holds and administers the copyright of `R` software and documentation.

`R` is an official part of the Free Software Foundation's GNU project, and the `R` Foundation has similar goals to other open source software foundations.

Among the goals of the `R` Foundation are the support of continued development of R, the exploration of new methodology, teaching and training of statistical computing and the organization of meetings and conferences with a statistical computing orientation.

The `R` Foundation, hopes to attract sufficient funding to make these goals realities.

Furthermore, `R` is a stimulating environment where ideas can be easily prototyped into software.

The first part of this manual deeply explores three key concepts of the `R` world: environments, functions and packages.

The second part examines the double nature of `R` both as functional and object oriented programming language.

The third part illustrates some tools to improve the efficiency and speed of our coding: debugging, profiling, parallel computation and Rcpp.

Chapter 2

How R works

2.1 The *‘’read parse evaluate”* loop

The *‘’read parse evaluate”* loop is at the core of the R system. Understanding this mechanism and being able to manage and control it is a key point for writing efficient R codes.

When we type R commands, two things happen:

- those characters are parsed into an R expression
- the expression is then evaluated by the internal evaluator

This process, known as the *‘’read parse evaluate”* loop, is internally performed by the R system but, the same process is available to the end user by mean of two functions: `parse()` and `eval()`
.

Effectively, when typing any `a <- 1`, what happens is

```
eval(parse (text = "a <-  1"))
```

The inner parse section returns an object of class `expression` and afterward the expression is evaluated by the evaluator.

When we call functions `eval()` and `parse()` directly, we generally pass character strings as arguments to the `parse()` function either from quoted text strings or external files

```
parse(text = "a <- 1")
```

```
## expression(a <- 1)
```

```
parse(file = './input.R')
```

Expressions objects are special language objects of class `expression` which contain parsed but unevaluated R statements. Parsed expressions are stored in an R object that can be explored as standard list objects.

```
expr <-  parse (text = "a <- 1")
class(expr)

## [1] "expression"

str(expr)

##   expression(a <- 1)

as.list(expr)

## [[1]]
## a <- 1

as.list(expr[[1]])

## [[1]]
## `<-`
##
## [[2]]
## a
##
## [[3]]
## [1] 1
```

and even manipulated as standard list objects

```
expr <-  parse (text = "1+2")
as.list(expr[[1]])

## [[1]]
## `+`
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] 2

expr[[1]][[3]] <- 7
as.list(expr[[1]])

## [[1]]
## `+`
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] 7
```

The evaluation part of the R program consists of passing the object resulting from parsing the current expression to the R evaluator.

The parsed expression is then evaluated by the function `eval()`.

```
eval(expr)
```

```
## [1] 8
```

and the result is returned.

Because of the way R works, evaluating expressions is, except few exceptions, about evaluating functions calls. This is clearly true when we call any a standard function in R as:

```
mean(x = 1:10)
```

but, this is also true when we write any assignment statement. In fact R translates:

```
x <- 0
```

into a function call:

```
`<-`(x, 0)
```

and even a conditional construct such as:

```
if ( pi > 0) 1 else  0
```

```
## [1] 1
```

translates into a call to a function

```
`if`( pi > 0 , 1 , 0)
```

The `parse-eval` mechanism has at least three exceptions: constants, names and promises:

Constants in R are evaluated into themselves. Any expression as:

```
1
```

```
## [1] 1
```

is the evaluation of a constant while

```
-1
```

```
## [1] -1
```

turns into a call to a function:

```
`-`(1)
```

More on how `R` evaluates function in the chapter dedicated to functions

A symbol is a variable name with a value associated to it: `x` is a symbol, or a symbol name:

```
x <- 0
class(quote(x))
```

```
## [1] "name"
```

Symbols in R may be made of lower or capital letters, numbers and the special characters `"."` and `"_"`. Almost any rule is a valid one when defining a symbol

```
x <- 0
x0 <- 0
x_0 <- 0
x.0 <- 0
.x <- 0
```

Standard symbols cannot start with a number or a `"_"`. Any name staring with a `"."` is a hidden name meaning that it is not returned by a call to `ls()` unless argument `all.names` is set to `TRUE`.

```
.x <- 0
ls()
```

```
## character(0)
```

```
ls(all.names = TRUE)
```

```
## [1] ".x"
```

When we ask `R` to evaluate a symbol, `R` looks for the value associated to that symbol, first in the current environment and, in case the symbol is not found within current environment, `R` looks progressively in all the parents environments until the object value is returned or an error occurs as the symbol is not found. This key idea will be fully discussed in the chapter dedicated to environments.

2.2 Assignment

When typing `a = 1` at the command prompt, the value `1` in assigned to the symbol `a`. The `=` operator is used to perform the assignment. In fact, `R` provides three operators for assignments: `=`, `<-` and `<<-` the last two being bi-directional.

Operators `=` and `<-` assign into the environment in which they are evaluated. Therefore, at the command prompt `a = 1` is equivalent to `a <- 1`.

When an assignment is done on formal parameter lists within functions calls, assignment is performed in the environment where the function is evaluated if the `=` operator is used while the same assignment occurs in the local environment in case the of the `<-` operator. As a simple example, we can consider a simple call to any function i.e. `median()`. First we clean up our workspace:

```
rm(list = ls())
```

Then we call `median()` using both `=` and `<-` operators for parameters assignement:

```
median(x = 1:10)

## [1] 5.5

exists("x")

## [1] FALSE

median(x <- 1:10)

## [1] 5.5

exists("x")

## [1] TRUE

x

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

In practice, the way the evaluator understands assignment is:

```
'='(a, 1)
a

## [1] 1
```

```
'<-'(b, 2)
b
```

```
## [1] 2
```

R supports multiple assignments with both operators.

```
x = xx = 1
y <- yy <- 0
c(x, xx, y, yy)
```

```
## [1] 1 1 0 0
```

Attention should be paid as:

```
k = p <- 0
c(k, p)
```

```
## [1] 0 0
```

works correctly but:

```
k <- p = 0
```

returns an error. This happens as

`k = p <- 0` translates to `'='(k,'<-'(p, 0))`

while

`k <- p = 0` is interpreted as `'='('<-'(k, p), 0)`

as the `<-` operator takes precedence on the `=` operator.

Finally, operator `<<-` is used to assign into the parent environment. As an example consider:

```
env <-new.env()
parent.env(env)
```

```
## <environment: R_GlobalEnv>
```

```
with(env, x <<- 8)
```

In this case the assignment `x = 8` is performed within the parent frame of `env`, that is `R_GlobalEnv`. Thus:

```
ls(env = env)
```

```
## character(0)
```

does not show any `x` symbol while `x` is still available in `env`:

```
get("x", env = env)
```

```
## [1] 8
```

as, since the evaluator does not find `x` in the local frame, it looks for `x` in the parent frame. In fact:

```
x
```

```
## [1] 8
```

2.3 Removing objects

To remove objects, the function `rm()` can be used. The function `remove()` may be considered as an alias for the `rm()` function.

As seen above, `ls()` returns a vector containing all objects in the current environment. To remove all objects in the current environment, all you need is

```
rm(list = ls())
ls()
```

```
## character(0)
```

Of course, the `list` argument can contain any character vector with object names.

```
x <- 1; y <- 2; z <- 3
ls()
```

```
## [1] "x" "y" "z"
```

```
rm(list = c("x", "y", "z"))
ls()
```

```
## character(0)
```

When argument are not already in a vector, they can be passed directly:

```
x <- 1; y <- 2; z <- 3
ls()
```

```
## [1] "x" "y" "z"

rm("x", "y", "z")
ls()

## character(0)
```

When arguments are passed directly, and not in the character vector `list`, it is not mandatory to quote them.

```
x <- 1; y <- 2; z <- 3
ls()

## [1] "x" "y" "z"

rm(x, y, z)
ls()

## character(0)
```

2.4 Garbage collection

When objects are no longer used, and this clearly happens when objects are deleted. `R` releases immediately the memory they filled in the system. This is done automatically by the garbage collector `gc()`.

We can call `gc()` to see how much memory `R` is using for allocating objects

```
gc()

##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 147432  7.9     350000 18.7   350000 18.7
## Vcells 338920  2.6     786432  6.0   669952  5.2
```

and as a proof, we can create `100x10^7` elements matrix

```
n <- 100*10^7
big_matrix <-  matrix(1:n, ncol = 100)
```

and check how much memory `R` is using:

```
gc()

##             used   (Mb) gc trigger   (Mb) max used   (Mb)
## Ncells    147450    7.9  3.500e+05   18.7  3.5e+05   18.7
## Vcells 500338959 3817.3  1.051e+09 8015.5  1.0e+09 7632.4
```

The increase of memory usage is related to the newly created matrix that takes:

```
print(object.size(big_matrix), units = "Gb")
```

```
## 3.7 Gb
```

When this matrix is removed, the memory is immediately released to the operating system.

```
rm(big_matrix)
gc()
```

```
##          used (Mb) gc trigger   (Mb) max used   (Mb)
## Ncells 147755  7.9     350000   18.7  3.5e+05   18.7
## Vcells 339529  2.6  840481003 6412.4  1.0e+09 7632.4
```

Chapter 3

Environments

3.1 Introduction

Assuming that we are all familiar with *classic* R objects such as *vectors*, *matrices*, *lists*, *data.frames*, etc …, this chapter takes into consideration a critical type of objects: **environments**.

Within the R computation mechanism, environments play a crucial role as they are constantly used by R just behind the scene of interactive computation.

An environment is an object that takes care of mapping variable names to values. Each mapping is called a binding.

Being able to understand and manage environments represents a key step in the R programming learning curve.

3.2 Environments in R

The *environment* definition is clearly stated by in *R Language Definition* manual:

Environments can be thought of as consisting of two things.

- A frame, consisting of a set of symbol-value pairs,
- an enclosure, a pointer to an enclosing environment.

Given that a frame is a set of objects each of them associated to a name, where a name is a simple character string, in practice, we can consider an environment as a self contained portion of memory containing a frame. Each environment can access one and only one other environment known as the parent environment.

Environments in R are created, and eventually destroyed, under many circumstances.

Any R session has an environment associated known as the *global environment.* as returned by functions `globalenv()` and `environment()`:

```
globalenv()
```

```
## <environment: R_GlobalEnv>
```

```
environment()
```

```
## <environment: R_GlobalEnv>
```

When we are working with `R` in interactive mode, we are using the frame within the `globalenv` as a *container* for our objects:

```
x <- 0
ls.str(globalenv())
```

```
## x :  num 0
```

Any package has at least one environment:

```
as.environment("package:stats")
```

```
## <environment: package:stats>
## attr(,"name")
## [1] "package:stats"
## attr(,"path")
## [1] "/usr/lib/R/library/stats"
```

Almost all functions have an environment as part of their definition:

```
environment(mean)
```

```
## <environment: namespace:base>
```

User defined functions have an environment too:

```
f <- function() NULL
environment(f)
```

```
## <environment: R_GlobalEnv>
```

Function `environmentName()` returns the name of an environment. As a result we may query `R` for the environment of function `f()`:

```
environmentName(environment(f))
```

```
## [1] "R_GlobalEnv"
```

or for the name of the environment associated to a package:

```
require("scuba", quietly = T)

## scuba 1.7-0
## Type 'help(scuba)' for an introduction
## Read the warnings in 'help(scuba.disclaimer)'
## For news about changes to the package, type 'news(package='scuba')'

environmentName(as.environment("package:scuba"))

## [1] "package:scuba"
```

Unfortunately, function `environmentName()` does not always return the expected results:

```
env <- new.env()
environmentName(env)

## [1] ""
```

Environment names for packages and namespaces are assigned at the `C` level. Therefore, user created environments do not reveal names. Users cannot set the name of an environment in `R` even through a, possibly misleadingly named, function called `environmentName()` exists. This function is really only meant for packages and namespaces, not other environments.

3.3 The ''environment tree structure"

The definition of environment also states that an environment is made of an enclosure: a pointer to an enclosing environment. As a consequence, any environment has a parent environment that, as an environment has a parent environment. This chain of parent environments, known as the *environment tree structure*, roots to a special environment called the *empty environment* that, as stated by its name, contains no objects.

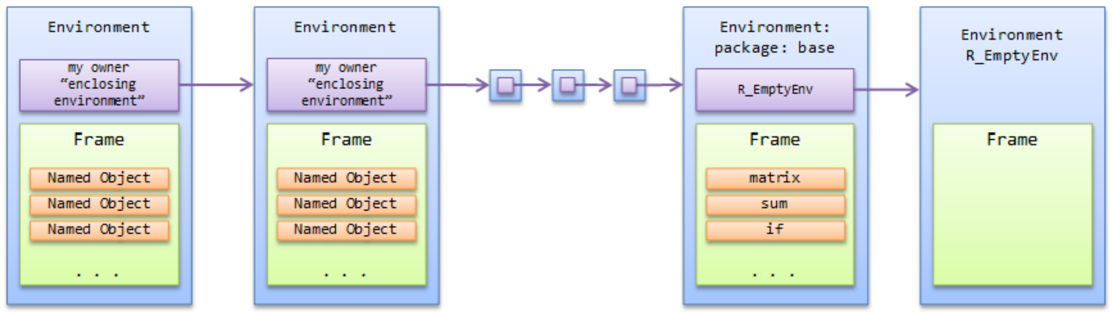


Figure 3.1: Figure from http://obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/

`R` has a very useful function, known as `parent.env()`, that returns the parent of any given environment:

```
parent.env(globalenv())
```

```
## <environment: package:scuba>
## attr(,"name")
## [1] "package:scuba"
## attr(,"path")
## [1] "/home/andrea/R/x86_64-pc-linux-gnu-library/3.1/scuba"
```

In order to visualize the environment tree structure we can easily define a function that returns this structure starting from any given environment:

```
tree <- function(env){
  cat("+ ", environmentName(env), "\n")
  if(environmentName(env) != environmentName(emptyenv())){
    env <- parent.env(env)
    Recall(env)
  }
invisible(NULL)
}
```

The above function make use of function `Recall()` that will be examined in the chapter dedicated to functions.

We can test `tree()` starting with `globalenv()` as argument:

```
tree(env = globalenv())
```

```
## +  R_GlobalEnv
## +  package:scuba
## +  package:knitr
## +  package:stats
## +  package:graphics
## +  package:grDevices
## +  package:utils
## +  package:datasets
## +  Autoloads
## +  base
## +  R_EmptyEnv
```

Or we may want to use the built in functions `search()` that returns similar results

```
search()
```

```
##  [1] ".GlobalEnv"        "package:scuba"     "package:knitr"
##  [4] "package:stats"     "package:graphics"  "package:grDevices"
##  [7] "package:utils"     "package:datasets"  "Autoloads"
## [10] "package:base"
```

When we attach a `list`, usually a `data.frame`, we actually insert an entry in the environment tree structure in the position given by the `pos` argument of function `attach()`. As this parameter defaults to `pos=2L`, most of the times we attach just underneath the global environment:

```
attach(data.frame(NULL))
search()
```

```
##  [1] ".GlobalEnv"        "data.frame(NULL)"  "package:scuba"
##  [4] "package:knitr"     "package:stats"     "package:graphics"
##  [7] "package:grDevices" "package:utils"     "package:datasets"
## [10] "Autoloads"         "package:base"
```

When loading libraries, functions `library()` or `require()` work on a similar basis and use the same parameter `pos = 2L`

```
library(MASS)
search()
```

```
##  [1] ".GlobalEnv"        "package:MASS"      "data.frame(NULL)"
##  [4] "package:scuba"     "package:knitr"     "package:stats"
##  [7] "package:graphics"  "package:grDevices" "package:utils"
## [10] "package:datasets"  "Autoloads"         "package:base"
```

3.4 How R looks for objects

When R looks for any object, a symbol value pair, by default R looks for a matching symbol in the current environment and, if a matching symbol is found, the corresponding value is returned.

In case we want to search starting from a different environment we are usually able to specify it directly. As an example, we may consider the well known function `get()` that has an argument `envir` specifying which environment to search, at least as a starting point.

As a result, we can create an object named `Formaldehyde` in the current environment:

```
Formaldehyde <- data.frame()
```

and use `get()` to find it along with the environment where to look for:

```
get("Formaldehyde", envir = globalenv())
```

```
## data frame with 0 columns and 0 rows
```

Note that an object with the same name exists in the environment of `package:datasets` and we can find it by specifying the right environment:

```
get("Formaldehyde", envir = as.environment("package:datasets"))
```

```
##   carb optden
## 1  0.1  0.086
## 2  0.3  0.269
## 3  0.5  0.446
## 4  0.6  0.538
## 5  0.7  0.626
## 6  0.9  0.782
```

When R does not find the required symbol in the current environment, R looks in the parent environment and then in the parent of the parent until R either finds the symbol in any environment or reaches the empty environment. In the latest case, as by definition the empty environment contains no objects, R returns an error.

Given this search mechanism, R stops searching as soon as it finds an object with the corresponding name ignoring any object with the same name in any other environment in the environment tree structure.

This effect, known as masking, may result in quite embarrassing situations.

As a very simple example, suppose we define a simple function for computing circumference length given radius as argument:

```
circumference <- function(radius) 2*pi*radius
```

and that, at any point of our working session we defined:

```
pi <- 0
```

The result we would gain looks quite embarrassing:

```
circumference(1)
```

```
## [1] 0
```

In this case the object pi in the globalenv() :

```
get("pi", envir = as.environment(globalenv()))
```

```
## [1] 0
```

masks the same symbol in the base environment

```
get("pi", envir = as.environment(baseenv()))
```

```
## [1] 3.142
```

A robust method that reduce the risk of masking consists in specifying the package we are calling objects from: We could achieve this goal by using the ‘‘::” operator:

```
circumference <- function(radius) 2*base::pi*radius
circumference(1)
```

```
## [1] 6.283
```

Finally, any conflict is returned by:

```
conflicts()
```

```
## [1] "Formaldehyde" "npk"          "pi"
```

3.5 Computing with Environments

As we have seen, environments are an essential components of the R working mechanism. As a consequence, it should not come as a surprise if environments are defined as R objects themselves.

As a consequence of being R objects, environments can be created:

```
env <- new.env()
```

and eventually deleted:

```
rm(env)
```

The `frame` component of an environment can be used as an objects place holder almost as we do with lists. We can place objects within an environment at least in three different ways:

by using the `$` operator:

```
env$zero <- 0
```

by using function `with()`:

```
with(env , one <- 1)
```

by using function assign:

```
assign("three", 3, envir  = env)
```

Finally, we can browse environment `env` with standard functions `ls()` or `ls.str()` to check our result:

```
ls(env)
```

```
## [1] "one"   "three" "zero"
```

```
ls.str(env)
```

```
## one :  num 1
## three :  num 3
## zero :  num 0
```

Suppose we want to store several objects at once into an environment, we may want to define a function `fill_envir()` that saves any series of objects within an environment:

```
fill_envir <- function(..., envir = globalenv()){
  this_list <- list(...)
  Map(function(...) assign(..., envir = envir) , names(this_list), this_list)
  invisible(NULL)
}
```

The above function takes `...` as argument and internally makes use of function `Map()` with an *anonymous function* as first argument. All this interesting concepts will be exhaustively explained in the next chapters.

By using function `fill_env()`, we can create a new environment and, subsequently, fill it with objects:

```
env1 <- new.env()
fill_envir(one = 1, seven = 7, envir = env1)
ls.str(env1)
```

```
## one :  num 1
## seven :  num 7
```

As we do with `list()`, we may also want a function `envir()` that directly creates an environment with named objects inside:

```
envir <- function(..., hash = TRUE, parent = parent.frame(), size = 29L){
  envir <- new.env(hash = hash, parent = parent, size = size)
  fill_envir(..., envir = envir)
  return(envir)
}
```

Note that, we have used the newly created function `fill_envir()` within the body of `envir()`; writing modular functions, reusable within new functions, is a key point for producing efficient `R` coding.

Function `envir()` is now ready to be used for creating new environments:

```
env2 = envir(six = 6, seven = 7)
ls.str(env2)
```

```
## seven :  num 7
## six :  num 6
```

Up to now we have noticed that environments behave very similarly to lists but, at this point of this explanation, we must point out at least three differences that exists between environments and lists:

First of all, within environment all objects must have a name while lists do not impose this restriction. In fact, we can create a list with unnamed components:

```
list (0, 1)
```

```
## [[1]]
## [1] 0
##
## [[2]]
## [1] 1
```

but we cannot do the same with environments nor using function `envir()`:

```
envir(0,1)
```

```
## Error: zero-length inputs cannot be mixed with those of non-zero length
```

or, any other approach that attempts to create nameless objects within an environment.

This sound quite logical as the definition of environment states that: *Environments consist of a frame, or collection of* ***named*** *objects.*

Similarly, we may have lists with duplicated components:

```
l <- list (x = 0 , x = 1)
l$x
```

```
## [1] 0
```

This idea may look strange but it is a basic example *masking* within `R`.

When we try to repeat the same experiment with environments, we may observe a different behavior:

```
env <- envir(x = 0, x = 1)
ls.str(env)
```

```
## x :  num 1
```

In this case, the second argument: `x = 1` simply reassigns a different value to `x`.

Finally, as opposite to lists, within environments, the order objects were placed in does not matter. The frame is a collection of named objects and only names matter. As a consequence, objects of an environment are always displayed in alphabetical order:

```
env <- envir(b = 2, a = 1)
ls.str(env)
```

```
## a :  num 1
## b :  num 2
```

The second part of the definition of environment states that an environment is made of an enclosure: a pointer to another environment.

As a consequence of this definition, when we create a new environment, it has, by definition, a parent environment.

Unless differently specified, the parent of the newly created environment is the environment where the environment was created.

As a result, if we create an environment, say `env0`, within the global environment, the latest results as the parent of `env0`.

```
env0 <- new.env()
parent.env(env0)
```

```
## <environment: R_GlobalEnv>
```

We can pass `env0` as an argument to the `tree()` function:

```
tree(env0)
```

```
## +
## +  R_GlobalEnv
## +  package:MASS
## +  data.frame(NULL)
## +  package:scuba
## +  package:knitr
## +  package:stats
## +  package:graphics
## +  package:grDevices
## +  package:utils
## +  package:datasets
## +  Autoloads
## +  base
## +  R_EmptyEnv
```

Note that, the name of the environment `env0` is not returned by function `environmentName()`. This happens as the name of an environment is stored into the underlying *C* function and no assignment or replacement method exist, at the moment, for environments.

We may even create an environment, say `env1` and specifically declare its parent environment:

```
env1 <- new.env(parent=baseenv())
```

Again, we can use function `tree()` to see the effects of the previous statement:

```
tree(env1)
```

```
## +
## +  base
## +  R_EmptyEnv
```

It should be clear at this point that the structure of the environment tree is a key element in the `R` programming mechanism.

These concepts will be very often recalled in the chapter dedicated to functions.

3.6 Copy on modify

When we do an assignment, `R` reserves a portion of memory for that object. We can *display* objects memory addresses by using a short function:

```
mem_add <- function(x) substring(capture.output(.Internal(inspect(x))), 2, 17)
```

beside its cryptic output, `mem_add()` allows us to verify that, given two different objects:

```
x <- 0
y <- 0
```

they have different memory address:

```
mem_add(x)
```

```
## [1] "2108658 14 REALS"
```

```
mem_add(y)
```

```
## [1] "e15a78 14 REALSX"
```

and that, given:

```
x <- 0
```

if we assign

```
y <- x
```

they share the same memory address:

```
identical(mem_add(x), mem_add(y))
```

```
## [1] TRUE
```

that is: when vector `x` is copied into vector `y`, both objects share the same memory address.

When existing objects are modified, usually `R` objects follow a *copy on modify* semantic; that is the object is copied into a different memory address.

In practice, given an object:

```
x <- 1:5
```

with its address

```
mem_add(x)
```

```
## [1] "11d9830 13 INTSX"
```

if we modify it

```
x[3] <- 0L
```

`R` modify its address too

```
mem_add(x)
```

```
## [1] "14a8960 13 INTSX"
```

If we apply the same concept to lists, given a list

```
list0 <- list(x = 0)
```

and its copy

```
list1 <- list0
```

both list share the same address

```
identical(mem_add(list0), mem_add(list1))
```

```
## [1] TRUE
```

but, if we modify `list1`

```
list1$x <- 1
```

`list0` and `list1` now have different addresses:

```
identical(mem_add(list0), mem_add(list1))
```

```
## [1] FALSE
```

This mechanism: *copy on modify*, allows to preserve the value of `list0` even if `list1` is modified.

Prior to `R 3.1` when modifying a list the entire list was copied. With version `3.1` we had a nice change that clearly helps in keeping memory usage under control.

Suppose we have two copied list made of more than one element:

```
list0 <- list(x = 1:100, y = rpois(100, 100))
list1 <- list0
```

and we modify only the second vector of the second list: `y`:

```
list1$y[1] <- 0L
```

We can now observe that the memory address is modified only for the second element of the list while `list0` and `list1` keep sharing the same address for the first vector: `x`

```
lapply(list0, mem_add)
```

```
## $x
## [1] "209b9f0 13 INTSX"
##
## $y
## [1] "1e0be70 13 INTSX"
```

```
lapply(list1, mem_add)
```

```
## $x
## [1] "209b9f0 13 INTSX"
##
## $y
## [1] "d6b9d0 13 INTSXP"
```

In conclusion, we could say that `R`, at leat in its newest versions, uses a *partial copy on modify* semantic.

The same semantic does not apply to environments, that is *environments do not copy on modify*:

As an proof of concept, we can create an environment with some objects in it:

```
env0 <- new.env()
env0$x <- 0
```

Afterward, we copy our newly created environment `env0` into a second environment, say `env1`.

```
env1 <- env0
```

As `env1` is a copy of `env0`, both environments contain the same symbols with the same values associated to them.

```
env0$x
```

```
## [1] 0
```

```
env1$x
```

```
## [1] 0
```

As environments do not copy on modify, if we now modify `x` within `env1`:

```
env1$x <- 1
```

We can easily observe that the value of `x` within `env0` is modified too:

```
env0$x
```

```
## [1] 1
```

The previous example clearly shows that any modification on `env1` also affects `env0`. This is possible as `env0` and `env1` share the same memory address even after the modification `env1$x <- 1`:

```
identical(mem_add(env0), mem_add(env1))
```

```
## [1] TRUE
```

3.7 Hashed environments

When we create a new environment, by setting `hash=TRUE`: the default value, we create a hashed environment.

In computer science, a hash table or hash map is a data structure that uses a hash function to map identifying values, known as keys, to associated values. Thus, a hash table implements an associative array. The hash function is used to transform the key into the index (the hash) of an array element (the slot or bucket) where the corresponding value is to be sought.

Hashed environment, allow *value look up by symbol* faster than traditional methods at the price of the hash table implementation.

As a proof of concept we may consider the following example.

First, we create a simple data frame whose rows represent `name-value` pairs:

```
options(stringsAsFactors = FALSE)
n = 10^6
df = data.frame(name = paste("p", 1:n, sep = "."), value = 1:n)
head(df,  3)
```

```
##   name value
## 1  p.1     1
## 2  p.2     2
## 3  p.3     3
```

Secondly, we create a new environment and we fill it with the *name-value* pairs so that we define, within the newly created environment, `n` objects of value `i` and name `p.i`:

```
env = new.env(hash = T)

system.time(
  Map(function(...) assign(..., envir = env), x = df$name, value = df$value)
)
```

```
##    user  system elapsed
##  27.435   0.116  27.554
```

As we can see, implementing the hash table require a certain amount of computing time.

We now define a random sample of names:

```
k = 100
what = paste("p", sample(1:n, k), sep ="." )
```

and finally, we want to create a vector `out` containing the values corresponding to each name. In practice, if we selected `what <- c(p.1,p.2,p.3)` we would like `R` to return `c(1,2,3)`.

In order to achieve this result we may use either a crazy `for` loop approach

```
out <-  numeric(k)
system.time({
for (i in 1:k){
  out[i] <-  df$value[df$name == what[i]]
}})
```

```
##    user  system elapsed
##   4.703   0.073   4.782
```

or the common `R` vectorized approach

```
system.time({df$value[is.element(df$name , what)]})
```

```
##    user  system elapsed
##   0.071   0.001   0.073
```

or, finally, the new hash approach

```
system.time({
  unlist(mget(what,  envir =   env))
})
```

```
##    user  system elapsed
##  22.507   0.064  22.574
```

Definitely, the hash approach, if we are willing to pay the computational price required for building the hash table, offers a clear advantage.

Chapter 4

Functions

4.1 Functions structure

When working with R we all make constant use of functions and, when developing, we create new functions so that functions look like very familiar R objects. Nevertheless, understanding the theory and the rationals underlying R functions may help to create much more efficient and possibly elegant coding.

We can create and assign functions to a variable names as we do with any other object:

```
f <- function(x, y = 0) {
  z <- x + y
  z
}
```

Eventually, we can delete any function with the usual call to `rm()` or `remove()`

Functions are objects with three basic components:

- a formal arguments list
- a body
- an environment.

```
formals(f)
```

```
## $x
##
##
## $y
## [1] 0
```

```
body(f)
```

```
## {
##     z <- x + y
##     z
## }
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

4.1.1 Formals

Formals are the formal arguments of a function returned as an object of class `pairlist` where a `pairlist` can be thought as something similar to a list with an important difference:

```
is.null(pairlist())
```

```
## [1] TRUE
```

```
is.null(list())
```

```
## [1] FALSE
```

that is: a `pairlist` of length zero is `NULL` while a `list` is not.

When we call a function, formals arguments can be specified by position or by name and we can mix positional matching with matching by name so that the following are equivalent:

```
mean(x = 1:5, trim = 0.1)
```

```
## [1] 3
```

```
mean(1:5, trim = 0.1)
```

```
## [1] 3
```

```
mean(x = 1:5, 0.1)
```

```
## [1] 3
```

```
mean(1:5, 0.1)
```

```
## [1] 3
```

```
mean(trim = 0.1, x = 1:5)
```

```
## [1] 3
```

Along with position and name, we can also specify formals by partial matching so that:

```
mean(1:5, tr = 0.1)
```

```
## [1] 3
```

```
mean(tr = 0.1, x = 1:5)
```

```
## [1] 3
```

would work anyway.

Functions formals may also have the construct `symbol = default`, that unless differently specified, forces any argument to be used with its default value.

Specifically, function `mean()` also have a third argument `na.rm` that defaults to `FALSE` and , as a result passing vectors with `NA` values to `mean()` returns `NA`

```
mean(c(1, 2, NA))
```

```
## [1] NA
```

While, by specifying `na.rm=TRUE` we get the mean of all non missing elements of vector `x`.

```
mean(c(1, 2, NA), na.rm = TRUE)
```

```
## [1] 1.5
```

The order `R` uses for matching formals against value is:

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Formals of a function are normally used within functions by the internal `R` evaluator but, we can use function `formals()` to expose formals explicitly.

```
formals(f)
```

```
## $x
##
##
## $y
## [1] 0
```

`args()` is an other function that displays the formals in a more user friendly fashion. Actually, `args(fun)` returns a function with the same arguments as `fun` but with an empty body.

```
args(f)
```

```
## function (x, y = 0)
## NULL
```

Surely, for programming purposes, `formals()` is a better choice as it returns a simple `pairlist` that can be handled as a list:

```
is.list(formals(mean))
```

```
## [1] TRUE
```

As a replacement method exists for function `formals`:

```
exists("formals<-")
```

```
## [1] TRUE
```

formals of a function can manipulated by using function `alist()`: a `list()` type function that handles unevaluated arguments

```
g <- function(x, y=0) x+y
g(1)
```

```
## [1] 1
```

```
formals(g)
```

```
## $x
##
##
## $y
## [1] 0
```

```
formals(g) <- alist(x=, y=1)
g(1)
```

```
## [1] 2
```

As an example of practical use of `formals()` we may decide to re-define function `mean()` that defaults `na.rm` to `TRUE` by simply:

```
formals(mean.default)$na.rm <- TRUE
mean(c(1,2,NA))
```

```
## [1] 1.5
```

Clearly, we now have copy of `mean.default()` in our `globalenv`:

```
exists("mean.default", envir = globalenv())
```

```
## [1] TRUE
```

Finally, let's notice that:

```
environment(mean.default)
```

```
## <environment: namespace:base>
```

remains the base environment: the environment where the function was created.

The "`...`" argument of a function is a special argument and can contain any number of `symbol=value` arguments . The "`...`" argument is transformed by `R` into a list that is simply added to the `formals` list:

```
h <- function (x, ...) {0}
formals(h)
```

```
## $x
##
##
## $...
```

The "`...`" argument can be used if the number of arguments is unknown. Suppose we want to define a function that counts the number of rows of any given number of data frames we could write:

```
count_rows <- function(...) {
  list <- list(...)
  lapply(list, nrow)
}

count_rows(airquality, cars)
```

```
## [[1]]
## [1] 153
##
## [[2]]
## [1] 50
```

Similarly, the “`...`” arguments becomes very handy when the “`...`” arguments will be passed on to another function as it often happened when calling `plot()` from within another function. The following example shows a basic plot function used for depths plotting where additional graphics parameters are passed via “`...`”:

```
time <-  1:13
depth <-  c(0,9,18,21,21,21,21,18,9,3,3,3,0)

plot_depth <-  function ( time , depth , type = "l", ...){
  plot(time, -depth, type = type,
       ylab = deparse(substitute(depth)), ...)
}
par(mfrow = c(1, 2))
plot_depth(time, depth, lty = 2)
plot_depth(time, depth, lwd = 4, col = "red")
```

4.1.2 Body of a function

The body of a function is a parsed R statement. In practice, this implies that the body of a function needs to be correct from a formal point of view but no evaluation of the body of a function occurred yet.

As a result, this function would return an error:

```
wrong <- function(x) {x =}
```

as its body is not a correct `R` statement.

While this function:

```
right <- function(x){x+y}
```

is accepted by `R` as is formally correct even thought, except under specific circumstances, will always return an error:

```
right(x = 2)
```

```
## Error: object 'y' not found
```

The body of a function, is usually a collection of statements in braces but it can be a single statement, a symbol or even a constant.

The body of function is an object of class `call`:

```
f <- function(x) {x+1}
class(body(f))
```

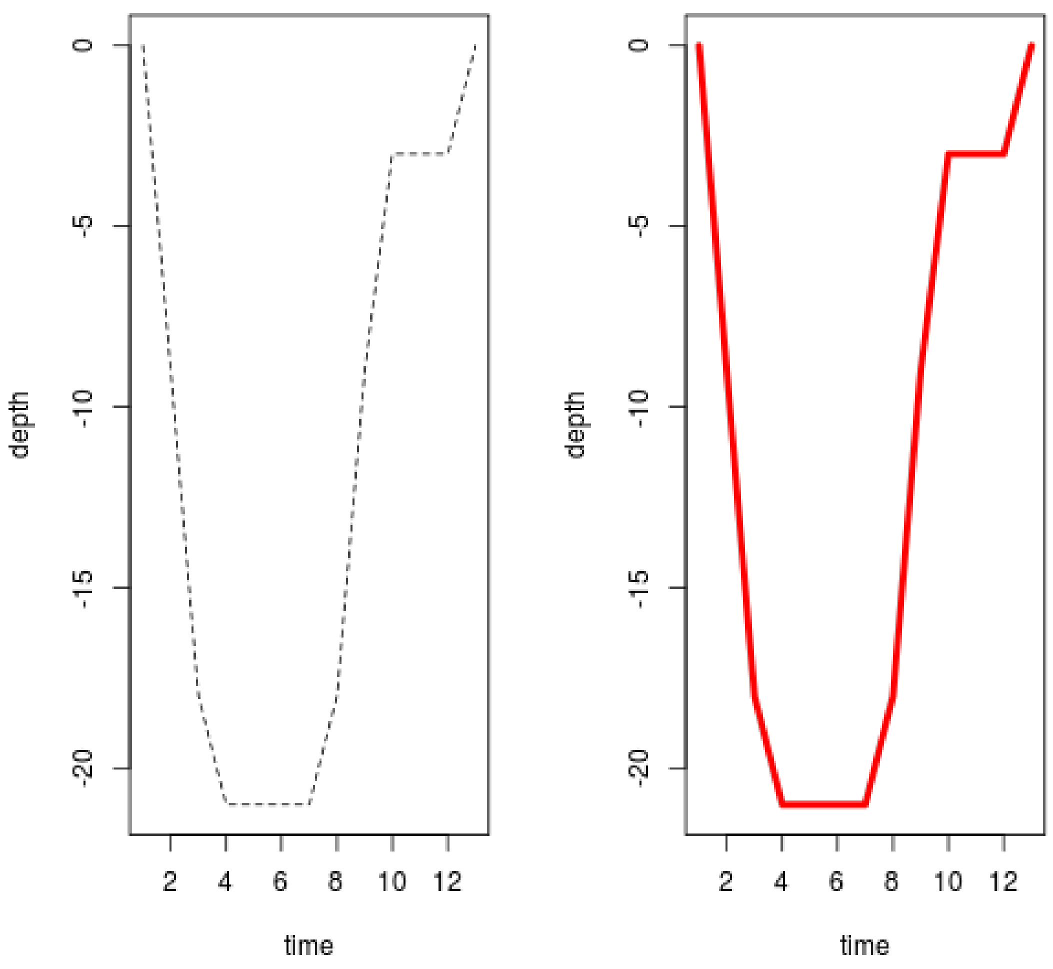


Figure 4.1: plot of chunk functions-017

```
## [1] "{"
```

and as a `call` object, the body of a function can be manipulated as a list:

```
as.list(body(f))
```

```
## [[1]]
## `{`
##
## [[2]]
## x + 1
```

and, as function `body()` has a replacement method: `body()<-`, the body of a function can be easily manipulated:

```
body(f)[[2]][[1]] <- `-`
f(1)
```

```
## [1] 0
```

This technique can be eventually used for testing *on the fly* small changes to a function without rewriting its full body.

4.1.3 Environment of a function

The environment of a function is the environment that was active at the time that the function was created. Generally, for user defined function, the Global environment:

```
f <- function(x){x+1}
environment(f)
```

```
## <environment: R_GlobalEnv>
```

or, when a function is defined within a package, the environment associated to that package:

```
environment(mean)
```

```
## <environment: namespace:base>
```

The environment of a function is a structural component of the function and belongs to the function itself.

As an example, we can define a function `f()` that simply returns zero

```
f <- function() 0
```

the environment of `f()` is clearly the `globalenv()`

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

we can modify the environment of a function and assign to `f()` a newly created environment

```
env <- new.env()
environment(f) <- env
environment(f)
```

```
## <environment: 0x3ae6070>
```

in case we delete environment `env`

```
rm(env)
```

`f()` will keep working

```
f()
```

```
## [1] 0
```

All of this happen as `env` and the environment of `f()` are two pointers to the same piece of memory address but they exist as separate objects.

As an example we may consider a function defined in a dedicated environment along with some other objects in the same environment.

```
env <- new.env()

with(env,{
     y <- 1
     g <- function(x){x+y}
     })


with(env, g(1))
```

```
## [1] 2
```

As we can see, clearly `g()` knows that `x=1` as it was passed to the function as an argument but, `g()` also remembers that `y=1` as `y` belongs to the environment `env`: the environment of `g()`.

The same behavior occurs many times when we develop `R` function and may lead to errors when calling these functions. Suppose we simply write:

```
y <- 1
g <- function(x){x+y}
g(2)
```

```
## [1] 3
```

The above example works as the environment of `g()` is now the global environment. But, as soon as we do:

```
rm(y)
```

clearly, `g()` will stop working as object `y` no longer exists in the global environment

```
g(1)
```

```
## Error: object 'y' not found
```

Notice that, if we define this odd function

```
f <- function() x
```

this function works if it finds variable `x` in its chain of searchable environments. As a result, if we define

```
env <- new.env()
env$x <- 0
environment(f) <- env
```

now `f()` returns zero as it finds `x` within its environment

```
f()
```

```
## [1] 0
```

if now delete `env`

```
rm(env)
```

`f()` will keep working

```
f()
```

```
## [1] 0
```

as a pointer to the same memory address exists as part of `f()` itself

Along with the environment where the function was created, functions usually interact with, at least, two more environments:

- The evaluation environment
- The calling environment

The evaluation environment is created any time the function is called. Within this environment, the formals arguments of the function are matched with the supplied arguments and the body of the function is evaluated.

The evaluation environment, as any other environment, has a parent. The parent of the evaluation environment of a function is the environment of the function. In other words, the function environment is the enclosure, the parent, of the evaluation environment.

As a proof of concept we can write simple function that returns the its evaluation environment along with the evaluated symbols that are created within this environment :

```
f <-  function(x){
  env <-  environment()
  env
}

env_f <- f(x = 0)
get("x", envir = env_f)
```

```
## [1] 0
```

As we can see, object `x` is bounded to the evaluation environment of `f()`.

The *calling environment* is the environment the function is called from. When using `R` interactively, the calling environment of a function is usually the global environment but, this is not always the case.

When we call a function, the function first looks for any variable in the evaluation environment and then in its enclosure; usually, for user defined functions, the global environment. In case no variable is found, `R` keeps searching along the environments stack until it reaches the empty environment. As we can see, this process does not take into account the calling environment.

When using `R` interactively, the environment of a function and the calling environment of that function often coincide: functions are defined in the global environment and called from the same environment.

In order to better understand the difference between the environment of a function and the calling environment of a function, we may consider a new environment, with a function `f()` defined in it, whose enclosure is forced to the `base` environment:

```
env <- new.env(parent = baseenv())
with(env, f <- function(x) {is.function(x)})
```

Function `f()` takes a single argument and returns `TRUE` in case it is a function, `FALSE` otherwise.

If we call this function with argument `x = c`:

```
with(env, f(c))
```

```
## [1] TRUE
```

`f()` returns `TRUE` as it is considering function `c()` from the `base` environment.

if we define an object `c` within environment `env`:

```
with(env, c <- 0)
```

and we call it:

```
with(env, f(x = c))
```

```
## [1] FALSE
```

now `f()` returns `FALSE` as it is considering variable `c` within the `env` environment and does not find function `c()` in the `base` environment.

If we now remove `c` from `env`:

```
remove(c, envir = env)
```

and we re-define `c` within our global environment:

```
c <- 0
```

when now calling `f(x = c)`,

```
with(env, f(x = c))
```

```
## [1] TRUE
```

we can see that `f()` now returns `TRUE` despite the `c <- 0` assignment in the global environment.

Basically, `f()` start searching from its environment: `env` and, if necessary, keeps searching along the environment tree structure that, in this case, does not include the `globalenv`.

`R` provides at least two useful functions to deal with the environments of a functions:

- `parent.env()`
- `parent.frame()`

`parent.env()` returns the environment in which the function was defined while `parent.frame(n = 1)` identify the environment from which the function was invoked.

In order to illustrate this concepts, we can define:

```
env_of_fun <- function(){
  evaluated_in <- environment()
  defined_in <- parent.env(evaluated_in)
  called_from <- parent.frame(n = 1)

  c(evaluated_in = evaluated_in,
    defined_in = defined_in,
    called_from = called_from)
}
env_of_fun()

## $evaluated_in
## <environment: 0x2826418>
##
## $defined_in
## <environment: R_GlobalEnv>
##
## $called_from
## <environment: R_GlobalEnv>
```

This function was defined in the global environment and called from the global environment.

Suppose we now define a new environment `env` and we move `env_of_fun()` in it:

```
env <- new.env()
env$env_of_fun <- env_of_fun
rm(env_of_fun)
```

when we now call `env_of_fun()`

```
with(env, env_of_fun())

## $evaluated_in
## <environment: 0x38f7308>
##
## $defined_in
## <environment: R_GlobalEnv>
##
## $called_from
## <environment: 0x2ca1fb8>
```

we can see that the calling environment is now different from the definition environment.

Understanding this idea can help to improve clarity and avoid annoying conflicts.

As an example, we can define function `f()` within a newly created environment `env` and use function `parent.frame()` within the newly created function:

```
rm(list = ls())
env <- new.env(parent = baseenv())
with(env, f <- function(x) {
  x <- eval(x, envir = parent.frame(n = 1))
  is.function(x)
  })
```

and observe that:

```
env$f(c)
```

```
## [1] TRUE
```

```
c <- 1
env$f(c)
```

```
## [1] FALSE
```

```
with(env, f(c))
```

```
## [1] TRUE
```

that is, function `parent.frame()` forced `f()` to look for `c` first inside the calling environment rather than the creation environment: `env` or its parent:

Similarly, in order to avoid conflicts between objects passed as arguments to a function and objects stored in any other environment, such as a package, we could define `f()` within `env` as:

```
env <- new.env(parent = baseenv())
with(env, f <- function(x) eval(x, parent.env(environment())))
```

in this case we can be sure that whenever we call `f()` it first looks for the value of `x` as stored either in `env` or its parent: :

Suppose, in fact, we call;

```
with(env, f(x = pi))
```

```
## [1] 3.142
```

```
pi <- 0
with(env, f(x = pi))
```

```
## [1] 3.142
```

we can observe that `f(x = pi)` always returns teh correct value for `pi`

4.2 Example: Remove all objects from the workspace

As an example of use of the environment of a function, we can consider several strategies to write a function capable of removing all objects from the `globalenv`. We can iniatially write a simple function:

```
clear = function(env = globalenv()) {
  obj = ls(envir = env)
  rm(list = obj, envir = env)
}
```

Function `clear()` removes all objects from a specified environment and seems to work correctly:

```
x <- 1; y <- 2; z <- 3
ls()
```

```
## [1] "c"     "clear" "env"   "pi"    "x"     "y"     "z"
```

```
clear()
ls()
```

```
## character(0)
```

At this point, should be obvious what is the drawback of this solution. Function `clear()` deletes also itself and, as a result, it cannot be reused without redefined it.

```
a <- 2
clear()
```

```
## Error: could not find function "clear"
```

This function can be improved, to keep function `clear()` when all other objects are deleted.

```
clear <-  function (env = globalenv()){
  objects <-  objects(env)
  objects <-  objects[objects != "clear"]
  rm(list = objects, envir = env)
  invisible (NULL)
}
```

Now the function can be used more than once.

```
a <- b <- c <- 0
clear()
a <- b <- c <- 1
clear()
```

Unfortunately, this function has also a drawback: it stops working when reassigned.

```
clean <- clear
rm (clear)
a <- b <- c <- 0
clean()
```

As defined above, function `clean()` also removes itself: only the object named `clear` is preserved.

```
a <- 3
clean()
```

```
## Error: could not find function "clean"
```

To dynamically keep function name, we may modify function clear as follow.

```
clear <- function (env = globalenv()){
  fname <- as.character(match.call()[[1]])
  objects <- objects(env)
  objects <- objects[objects != fname]
  rm(list <- objects, envir = env)
  invisible (NULL)
}
```

Nevertheless, beside the above solution, a smart way to obtain the same result is the follow:

```
assign("clean",
  function(env = globalenv()){
    rm(list = ls(envir = env), envir = env)
  },
  envir = attach(NULL, name = "myenv", pos = 2)
)
```

Through function `assign()`, function `clear()` is created in a new environment called `myenv`. In this way, all objects in the global environment can be removed without deleting function `clear()`

```
a <- b <- c <- 0
ls()
```

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

```
clean()
ls()
```

```
## character(0)
```

search()

4.3 Return Value

The last object called within a function is returned by the function and therefore available for assignment. Functions can return only a single value but, in practice, this is not a limitation as a list containing any number of objects can be returned.

Objects can be returned `visible` or `invisible`. This option has no effect on the assignment side but affects the way results are displayed when the function is called.

```
g <-  function (n){
 out <- runif(n)
 cat(head(out))
 invisible(out)
}

x <-  g(10^5)
```

```
## 0.3526 0.3616 0.3935 0.1849 0.8523 0.5663
```

```
length(x)
```

```
## [1] 100000
```

Sometimes, we may want a function that does any job but returns nothing. In this case, the return value will be set to `NULL` and returned as invisible.

Suppose we need a function that `cat()` a message we can write:

```
msg <- function(x){
  cat(x, "\n")
  invisible(NULL)
}
```

and use it as:

```
msg("test message")
```

```
## test message
```

with no assignment nor returned value.

4.4 Operators

Operators in R are simple function. Specifically, operators are *infix* functions as opposite to standard functions that are defined as *prefix* as the name of the function comes before its arguments. Operators can be defined as function with the only constrain that their name must be surrounded with ''%". As a result, a simple operator that concatenate strings can be defined as:

```
"%+%" = function(x,y){paste(x, y, sep = "")}
"we " %+% "love " %+% "R !"
```

```
## [1] "we love R !"
```

A more complex approach, based on `R` capabilities as an object oriented programming language, takes advantage of, `+` being a generic function:

```
methods(`+`)
```

```
## [1] +.Date   +.POSIXt
```

As a result, different methods for generic function `+` can be defined for different classes of objects.

As an example, we may define a class of objects named `string`:

```
string <- function(x) {
  s <- as.character(x)
  class(s) <- "string"
  s
}
```

with a `+` method that concatenates strings:

```
`+.string` <- function(s1, s2) paste(s1, s2, sep = "")
```

and as a result:

```
a <- string("Mickey")
b <- string("Mouse")
a+b
```

```
## [1] "MickeyMouse"
```

4.5 Lazy evaluation

Functions arguments, except few exceptions, are, by default, *lazy*; that is, they are not evaluated when the function is called but only when the argument are explicitly used.

Let's take as an example this simple function where the `y` argument is never evaluated within the function body:

```
rm(list = ls())
f = function(x, y){
  x+1
}
```

We can call `f()` and pass a non existing object `z` to argument `y`. Clearly, this kind of statement would result in a error as `z` is not defined but, it works as a function argument:

```
f(x = 0, y = z)
```

```
## [1] 1
```

As we can see, `y` is assigned to `z` and `z` does not exit but, `R` does not return any error. This is because `y = z` is never evaluated within the function body.

As a second example we can consider this basic function that simply prints its arguments:

```
h <-  function(a , b){
  cat ("a is:", a, "\n")
  cat ("b is:", b, "\n")
  invisible(NULL)
}
```

If we call `h()` without passing any vale to `b` we see that:

```
h(a = "we love R")
```

```
## a is: we love R
```

```
## Error: argument "b" is missing, with no default
```

that is: `h()` returns an error only when the evaluation of `b` is required. Prior to that, this function works perfectly.

Usually, whenever a function returns an error if any argument is not provided and not yet evaluated, this is because a control mechanism has been programmed within the function body:

```
g <-function(x, y){
  call <- match.call()
  args <- match(c("x", "y"), names(call))
  if(any(is.na(args))) stop("All args must be provided!")
  pi
}

g(y = 1)
```

```
## Error: All args must be provided!
```

More formally, an unevaluated argument is called a `promise`. A promise is an object made of three slots:

- a value

- an expression
- an environment

Practically, when a function is called, any argument is associated to a promise object along with the expression associated to that argument and a pointer to the environment where the expression will be, eventually, evaluated and assigned to the argument symbol.

Evaluation of an argument is required when:

- Interfacing with foreign language
- Selecting a method for a generic function
- An argument needs to be assigned within a function

There is generally no way within `R` to check whether an object is a promise or not, nor is there a way to determine the environment of a promise.

Lazy evaluation permits flexible handling of missing arguments and computations depending on the expression for the argument rather than its value.

The following example is a good case in point:

```
rescale = function(x, location = min(x), scale = max(y)){
  y <- x-location
  y/scale
}
rescale(1:4)
```

```
## [1] 0.0000 0.3333 0.6667 1.0000
```

This function scales any vector, by default, in the `[0,1]` range.
Argument `scale` depends on the value of `y` that is not defined but, it will be defined: `y = x-location` prior to its evaluation `y/scale`.

Function `delayedAssign()` offers a direct mechanism for accessing promise mechanism outside a function

```
delayedAssign("promise" , {x+y})
x <- 0
y <- 1
eval(promise)
```

```
## [1] 1
```

4.6 Functions call

Functions in `R` can be called *directly* or by mean of a second function such as `do.call()` by passing a string corresponding to the function name.

4.6.1 do.call()

Function `do.call()` takes as input two arguments:

- either a function or a non-empty character string naming the function to be called.
- a list of arguments to the function call

Basically:

```
mean(x = 1:100, trim = 0.2)
```

corresponds to:

```
do.call("mean", list(x = 1:100, trim = 0.2))
do.call(mean, list(x = 1:100, trim = 0.2))
```

4.6.2 Example: Maximumum Likelihood Estimamates

As an example, we may consider a maximum likelihood estimator for normal distributions:

```
mle = function(theta, x){
  ml = function(theta, x) {
    ml = dnorm(x = x, mean = theta[1], sd = theta[2])
    ml = -sum(log(ml))
    }
    optim(theta, ml, x = x)$par
}
mle(theta = c(0, 1), x = rnorm(100, 5, 2))
```

```
## [1] 4.759 1.979
```

We can re-implement the estimator by using `do.call()`:

```
mle = function(theta, x){
  ml = function(theta, x) {
    ml = do.call(dnorm, list(x, theta[1], theta[2]))
    ml = -sum(log(ml))
  }
  optim(theta, ml, x = x)$par
}

mle(theta = c(0, 1), x = rnorm(100, 5, 2))
```

```
## [1] 5.252 2.060
```

The distribution name can be passed as an argument to the `mle()` and, as a consequence, to `do.call()` at the cost of a minor modification to the internal function `mle()`.

```
mle = function(theta, x, dist){
  dist = paste("d", dist , sep = "")
  ml = function(dist , theta, x) {
    ml = do.call(dist, list(x, theta[1], theta[2]))
    -sum(log(ml))
  }
  optim(theta,  ml, dist = dist  , x = x)$par
}
mle(dist = "norm" , theta = c(0, 1), x = rnorm(10, 5, 2))
```

```
## [1] 4.758 1.644
```

Now it works with most of two parameters distributions assuming that the right initial `theta` is provided.

```
mle(dist = "lnorm" , theta = c(0,1), x = rlnorm(100, 3, 1))
```

```
## [1] 3.0405 0.9058
```

```
mle(dist = "weibull" , theta = c(1,1), x = rweibull(100, 3, 1))
```

```
## [1] 3.458 1.017
```

Clearly, this is a good value generalization given the programming effort required.

4.6.3 `match.call()`

Function `match.call()` is used within functions and it simply returns the call that has been passed to a function

```
f = function(a, b){
  call = match.call()
  call}

my_call = f(2, 3)
my_call
```

```
## f(a = 2, b = 3)
```

```
class(my_call)
```

```
## [1] "call"
```

Any `call` is an object of class `call` that can explored as a list object:

```
my_call_list <- as.list(my_call)
my_call_list
```

```
## [[1]]
## f
##
## $a
## [1] 2
##
## $b
## [1] 3
```

`Call` objects can also be manipulated as list.

```
my_call$a <- 0
eval(my_call)
```

```
## f(a = 0, b = 3)
```

4.6.4 Example: Function `anyway()`

As an example, we consider a function with two arguments `a, b` that returns, in case both arguments are numeric, the sum of the arguments; the character variable `"a+b"` otherwise.

```
anyway = function(a , b){
  call <-  match.call()
  if (is.numeric(a) & is.numeric(b)) {call[[1]] <- as.name("sum")}
    else {
      call[[1]] <- as.name("paste" )
      call$sep <- "+"
    }
eval(call)
}

anyway(3, 6)
```

```
## [1] 9
```

```
anyway("c", 2)
```

```
## [1] "c+2"
```

4.6.5 Example: Function `write.csv()` revisited

As a as second application of `do.call()` we consider `write.csv()`. This function is a wrapper to `write.table()` forcing `sep = ","` and `dec = "."`.

Such a function could be easily written as:

```
write.csv <-  function(...) write.table(sep = ",", dec = ".", ...)
siris <- head(iris, 3)
write.csv(siris)
```

```
## "Sepal.Length","Sepal.Width","Petal.Length","Petal.Width","Species"
## "1",5.1,3.5,1.4,0.2,"setosa"
## "2",4.9,3,1.4,0.2,"setosa"
## "3",4.7,3.2,1.3,0.2,"setosa"
```

Nevertheless, if we try to pass any of the `sep` or `dec` arguments via the ‘’`...`” argument, the function returns error:

```
write.csv(siris, sep = ";")
```

```
## Error: formal argument "sep" matched by multiple actual arguments
```

Basically, the ‘’`...`” argument may take any argument to be passed to `write.table()` except `sep` and `dec`. In case any of these arguments is explicitly passed to the function, they have to be forced to the desired default values: `","` and `"."`.

A simplified version of `write.csv()` can be re-written as:

```
write.csv = function(...){
  call = match.call()
  call[[1]] = as.name("write.table")
  call$sep = ","
  call$dec = "."
  eval(call)
}

write.csv(siris, sep = ";")
```

```
## "Sepal.Length","Sepal.Width","Petal.Length","Petal.Width","Species"
## "1",5.1,3.5,1.4,0.2,"setosa"
## "2",4.9,3,1.4,0.2,"setosa"
## "3",4.7,3.2,1.3,0.2,"setosa"
```

Now, `sep = ";"` is simply ignored.

4.7 Recursive functions

A recursive function use recursion and can call itself until a certain condition is met.

As an example we may consider a function that takes `x` as an argument and keep dividing it by `2` until the result is greater than `2`. This idea can be implemented by a simple `while()` loop:

```
one_c <- function(x){
  while (x > 2){
    x <- x/2
}
x
}
one_c(10)
```

```
## [1] 1.25
```

or alternatively by the use of function `Recall()`: a placeholder for the name of the function in which it is called. It allows the definition of recursive functions which still work after being renamed

```
one_r <- function(x){
  if (x > 2 ){
   x <- x/2
   x <- Recall(x)
  }
x
}
one_r(10)
```

```
## [1] 1.25
```

in this case `Recall(x)` is equivalent to `one_r(x)`.

The use of recursion my look redundant in this simple example but, it given an idea of how much a function can change by the simple introduction of this concept.

When dealing with more complex problem, the use of recursion may help indeed to simplify our coding.

4.7.1 Example: Quicksort

A good example of the advantages, and possibly disadvantages, of using recursive function is represented by the implementation of the *quicksort*: a divide and conquer algorithm that first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

As a simple implementation we may consider:

```
quick_sort_r  <- function(x) {

  if(length(x) > 1) {
    base <- x[1]
    l <- Recall(x[x < base])
    m <- x[x == base]
    h <- Recall(x[x > base])

    c(l, m, h)
  }
  else x
}
```

that results in:

```
quick_sort_r(sample(1:10))
```

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

Note that its non recursive implementation could be:

```
quick_sort_c <- function(x , max_lev = 1000) {
  n <- length(x)
  i <- 1
  beg <- end <- max_lev
  beg[1] <- 1
  end[1] <- n+1

  while (i>=1) {
    L <- beg[i]
    R <- end[i]-1
    if (L<R) {
      piv <- x[L]
      if (i == max_lev)
        stop("Error: max_lev reached");

      while (L<R) {
        while (x[R]>=piv && L<R){
          R <- R-1
        }
        if (L < R){
          x[L] <-  x[R]
          L <- L+1
        }
        while (x[L]<=piv && L<R){
          L <- L+1
        }
        if (L<R) {
          x[R] <- x[L]
          R <- R-1
        }
      }
      x[L] <- piv
      beg[i+1] <- L+1
      end[i+1] <- end[i]
      end[i] <- L
      i <- i+1
    }
    else {
      i <- i-1
    }
  }
  return( x)
}
```

that keeps working

```
quick_sort_c(sample(1:10))
```

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

but does not express the same level of clarity.

Moreover, when looking for performances, the use of recursion in `R` is a clear advantage:

```
x <- sample(1:10^5)
system.time(quick_sort_r(sample(x)))
```

```
##    user  system elapsed
##    0.50    0.01    0.51
```

```
system.time(quick_sort_c(sample(x)))
```

```
##    user  system elapsed
##   2.910   0.052   2.967
```

4.7.2 Example: Left join

Suppose we want to implent a left join between three data frames:

```
df1 <- data.frame(id = 1:6, x1 = 1:6)
df2 <- data.frame(id = 2:4, x2 = 2:4)
df3 <- data.frame(id = 3:5, x1 = 3:5)
```

we will have to acheive this goal in two steps:

```
df12 <- merge(df1, df2, by = "id", all.x = T)
df123 <- merge(df12, df3, by = "id", all.x = T)
df123
```

```
##   id x1.x x2 x1.y
## 1  1    1 NA   NA
## 2  2    2  2   NA
## 3  3    3  3    3
## 4  4    4  4    4
## 5  5    5 NA    5
## 6  6    6 NA   NA
```

In case we have to repeat this task several times, expecialy with a variable number of data frames, we could define function `left_join()` as:

```
left_join <- function(df_list, by, all.x = T){
  df_merged <- merge(df_list[[1]], df_list[[2]], by = by, all.x = all.x)
  df_list <- df_list[-1]
  df_list[[1]] <- df_merged
  if (length(df_list) > 1){
    df_merged <- Recall(df_list, by = by, all.x = all.x)
  }
  df_merged
}
```

and use it as:

```
left_join(list(df1, df2, df3), by = "id")
```

```
##   id x1.x x2 x1.y
## 1  1    1 NA   NA
## 2  2    2  2   NA
## 3  3    3  3    3
## 4  4    4  4    4
## 5  5    5 NA    5
## 6  6    6 NA   NA
```

4.8 Replacement functions

Given any `f()` function sometimes we are allowed to write expressions like: `f(x) <- y`. For example, given any `data.frame`:

```
df <-  data.frame(x = 1:3, y = 3:1)
```

we can query for the names of the variables within the data.frane by:

```
names(df)
```

```
## [1] "x" "y"
```

in order to replace variables names, we often use:

```
names(df) <-  c("xx", "yy")
names(df)
```

```
## [1] "xx" "yy"
```

This is possible as a function `names<-()` exists and it is known as the replacement method for `names()`.

```
get("names<-")
```

```
## function (x, value)  .Primitive("names<-")
```

In principle any replacement function takes the general form of: `"f<-"(x, value)` with `value` being the replacement argument.

4.8.1 Example: Trim and replace

As an example, we may consider function `trim()` that trims any vector at a the quantile corresponding to the `p` (probability) argument:

```
trim <-  function(x, p){
  x[x <= quantile(x, p)]
}

trim(1:10, p = .25)
```

```
## [1] 1 2 3
```

A simple replacement method for this function can be written as:

```
"trim<-" <-  function (x, p, value){
  x[x <= quantile(x, p)] <-  value
  x
}
```

and can be used as:

```
y <- 1:10
trim(x = y, p = .25) <-  0
y
```

```
##  [1]  0  0  0  4  5  6  7  8  9 10
```

4.8.2 Replacing non assigned objects

Note that using replacement functions requires that the object passed as argument `x` exists in the calling environment of the function. As a proof of concept we can see that:

```
df <- data.frame(x = 0, y = 1)
names(df) <- c("a", "b")
```

works normally, while

```
names(data.frame(x = 0, y = 1)) <- c("a", "b")
```

```
## Error: target of assignment expands to non-language object
```

does not work as the `data.frame` to be modified does not exist.

Chapter 5

Functional Programming

5.1 R as a functional programming language

`R` can be considered as a functional programming language as it focuses on the creation and manipulation of functions and has what's known as first class functions.

In computer science, functional programming is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

Functional programming emphasizes functions that produce results that depend only on their inputs and not on the program state

In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function `f()` twice with the same value for an argument `x` will produce the same result `f(x)` both times. `R` is clearly a functional programming language.

Understanding the functional nature of `R` may help to improve clarity and avoid redundancy.

We will examine:

- First Class Functions
- Functions Closures
- Functions Factories
- Anonymous Functions
- Lists of Functions
- Functionals

5.2 First class functions

First-class functions are a key component of functional programming style.

A programming language is said to have first-class functions when the language supports:

- passing functions as arguments to other functions
- creating anonymous functions
- returning functions as the values from other functions
- storing functions in data structures.

and R has has first-class functions indeed.

In this example we pass function: `identity()` as argument to function `lapply()`

```
lapply(0, identity)
```

```
## [[1]]
## [1] 0
```

Here we make use of an anonymous function:

```
(function(x) sd(x)/mean(x))(x = 1:5)
```

```
## [1] 0.527
```

We can easily define a function that return a function

```
f <- function(){
  function(x) sd(x)/mean(x)
}
```

Finaly we store functions within a list:

```
function_list <- list(mean , sd)
```

5.3 Functions closures

A **function closure** or **closure** is a function together with a referencing environment.

Almost all functions in R are closures as they remember the environment where they were created. Generally, but not always, the global environment:

```
f <- function(x) 0
environment(f)
```

```
## <environment: R_GlobalEnv>
```

or the package environment

```
environment(mean)
```

```
## <environment: namespace:base>
```

Functions that cannot be classified as *closures*, and therefore do not have a referencing environment, are know as *primitives*. These are internal `R` function calling the underlying `C` code. `sum()` and `c()` are good cases in point:

```
environment(sum)
```

```
## NULL
```

As functions remember the environments where they were created, the following example does not return any error:

```
y <- 1
f <- function(x){x+y}
f(1)
```

```
## [1] 2
```

This is possible as `f()` is declared within the global environment and therefore `f()` remembers all objects bounded to that environment (the referencing environment), `y` included.

When we call a function, a new environment is created to hold the function's execution and, normally, that environment is destroyed when the function exits. But, if we define a function `g()` that returns a function `f()`, the environment where `f()` is created is the execution environment of `g()`, that is, the execution environment of `g()` is the referencing environment of `f()`. As a consequence, the execution environment of `g()` is not destroyed as `g()` exits but it persists as long as `f()` exists. Finally, as `f()` remembers all objects bounded to its referencing environment, `f()` remembers all objects bounded to the execution environment of `g()`

With this idea in mind, we can use the referencing environment of `f()`, that is the execution environment of `g()`, to hold any object and these objects will be available to `f()`.

```
g <- function(){
  y <- 1
  function(x) {x+y}
}
f1 <- g()
f1(3)
```

```
## [1] 4
```

Moreover, as `f()` is created within `g()` any argument passed to `g()` will be available to `f()` in its later executions.

```
g <- function (y) {
    function(x) x+y
}
f1 <- g(1)
f1(3)
```

```
## [1] 4
```

As a proof of concept, we may temporaly modify function `g()` in order to print the execution environment of `g()`

```
g_tmp <- function(y){
  print(environment())
  function(x) {x+y}
}
```

Than use `g()` to produce `f()`

```
f1 <- g_tmp(1)
```

```
## <environment: 0x1837860>
```

and finaly ask `R` for the environment associated with `f()`

```
environment(f1)
```

```
## <environment: 0x1837860>
```

As we can see, the execution environment of `g_tmp()` corresponds to the environment associated to `f()`.

Finally,

```
get("y" , env = environment(f1))
```

```
## [1] 1
```

shows where `y` is stored.

Notice that each call to `g()` returns a function with its own referencing environment:

```
f2 <- g(1)
environment(f1)
```

```
## <environment: 0x1837860>
```

```
environment(f2)
```

```
## <environment: 0xb08e88>
```

The referencing environments for `f1()` and `f2()` are different, despite that `f1()` and `f2()` are both returned by `g()`.

5.4 Functions Factories

In practice, we can use closures to write specific functions that, in turn, can be used to generate new functions. This allows us to have a double layer of development: a first layer that is used to do all the complex work in common to all functions and a second layer that defines the details of each function.

5.4.1 Example: A basic case in point

We can think of a simple function `add(x,i)` that add the value `i` to the value `x`. We could define this function as:

```
add <- function(x, i){
  x+i
}
```

Alternatively, we may consider a set of functions, say `f1(x)`, `f2(x)`, `...`, `fi(x)`, `...`, `fn{x}` that add `1,2,...,i,...,n` to the `x` argument. Clearly, we do not want to define all these functions but we want to define a unique function `f(i)`:

```
f <- function(i){
  function(x) {x+i}
}
```

capable of generating all `fi(x)` functions:

```
f1 <-  f(1)
f1(3)
```

```
## [1] 4
```

```
f2 <- f(2)
f2(4)
```

```
## [1] 6
```

In this simple example, this approach shows no benefit and possibly increases the complexity of our codes but, for more structured cases, it is definitely worth.

5.4.2 Example: MLE functions

As a more structured example, we may consider the development of a set of functions: `lnorm(x)`, `lweibull(x)`, `...` that compute max likelihood estimates for those distributions given a vector of data `x`:

```r
new_estimate <-  function(dist){
  estimate <-  function(x, theta){
    neglik <-  function(theta = theta , x = x, log = T){
      args <-  c(list(x), as.list(theta), as.list(log))
      neglik <-  -sum(do.call(dist,  args))
      neglik
    }
    optim(par = theta, fn = neglik , x = x)$par
  }
estimate
}
```

`new_estimate` returns a second function: `estimate()` whose body depends on the argument `dist` passed to `new_estimate()`.

Within `estimate()` we first define a third function `neglik()` and secondly, we minimize it within `optim()`.

The returned function: `estimate()` can be used as a generator of maximum likelihood estimation functions for any distribution as long as its corresponding `ddist()` exists in `R`.

Once we have `new_estimate()`, we can use it to define any MLE estimation function as long as its density function is defined. That is, we can now write a `llnorm()` that computes log-normal maximum likelihood estimates as simply as:

```r
llnorm <- new_estimate("dlnorm")
x <- rlnorm(100, 7 , 1)
llnorm(x, theta = c(mean(log(x)), sd(log(x))))
```

```
## [1] 7.0571 0.9919
```

and similarly:

```r
lweibull <- new_estimate("dweibull")
w <- rweibull(100, 2 , 1)
lweibull(w, theta = c(mean(w), sd(w)))
```

```
## [1] 1.985 1.028
```

5.4.3 Example: moving statistics

As a further example of functions factories we may consider a function `moving(f)` that returns `moving_f()` where `f()` could be `mean()`, `median()` or any other statistical function as long it returns a single value.

As a first step we may consider a simple function `g()` that returns the value of `f()` for any backward window of length `n` starting at `i`:

```
g <- function(i , x, n , f, ...) f(x[(i-n+1):i], ...)
g(i = 5 , x = 1:10,n = 3  , f= mean)
```

```
## [1] 4
```

```
g(i = 5 , x = 1:10,n = 3  , f= sd)
```

```
## [1] 1
```

Note that `g()` takes, among its inputs, a second function `f()` and apply it to the window `[(i-n+1):i]` of `x`.

As a second step we define a function `moving(f)` that takes any function `f()` as an input and define function `g()` within its body.

```
moving <- function(f){
  g <- function(i , x, n , f, ...) f(x[(i-n+1):i], ...)
  h <- function(x, n, ...) {
    N <- length(x)
    vapply(n:N, g, x , n , f, FUN.VALUE = numeric(1), ...)
  }
return(h)
}
```

Function `moving()` returns function `h()` that, in turn can be used to generate any `moving_f()` functions:

Function `vapply()` within `h()` is a *functional* used as a for loop replacement that will be fully explored when discussing *functionals*.

```
moving_average <- moving(mean)
moving_average(x = rpois(10, 10), n = 3)
```

```
## [1]  8.000  6.667  6.667  9.000 12.000 14.000 15.333 12.333
```

Eventually, argument ‘‘`...`” can be used to pass extra arguments to `f()`.

```
moving_average(x = rpois(10, 10), n = 3, trim = .5)
```

```
## [1]  9 11 12 12  9  9  9  9
```

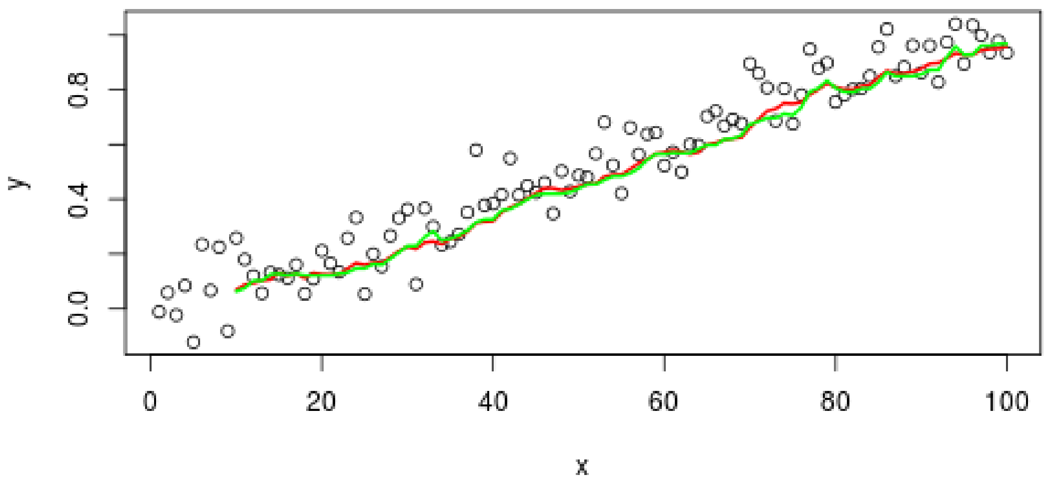
If necessary, function `moving()` ca be used in the form of anonymous function:

```
moving(sd)(rpois(10, 10), n = 5)
```

```
## [1] 2.074 1.871 1.643 1.817 1.817 2.074
```

Finally:

```
x <- 1:100
y <- seq(along = x, from = 0 , to = 1)+rnorm(length(x), 0 , .1)
plot(x, y)
lines(x[10:length(x)], moving(mean)(y, 10), col = "red", lwd = 2)
lines(x[10:length(x)], moving(median)(y, 10), col = "green", lwd = 2)
```



### Example: Truncated density function

Density function in R are usually specified by the prefixes `d` followed by a standard suffix for each ditribution. `dnorm()`, `dlnorm()`, `dweibull()`, etc …

Therefore, we use to write:

```
x <- c(7,10,13)
dlnorm(x , meanlog = 2, sdlog = 1)
```

```
## [1] 0.05691 0.03811 0.02616
```

in order to get density values at `x` from a lognormal distribution with parameters `2` and `1`.

In case we need value from a truncated distribution, as far as we know, we need to load an extra package such as `truncdist`. The package itself works perfectly. In fact, assuming that a `dlnorm()` function exists, we can get density values from a left truncated lognormal distribution with parameters `meanlog = 2` and `sdlog = 1` by simply writing:

```
require(truncdist)
```

```
## Loading required package: truncdist
## Loading required package: stats4
## Loading required package: evd
```

```
dtrunc(x, spec = "lnorm", a = 5)
```

```
## [1] 0.15963 0.05238 0.02128
```

where `a = 5` represents the left threshold for truncation

Nevertheless, the above command require a change in our programming style.

In principle, we would like to be able to write:

```
tdlnorm(x, meanlog = 2, sdlog = 1, L = 5)
```

where `L = 5` represents the left threshold for truncation

so that we could have the same programming style, just with different parameters, for both truncated and not truncated distribution.

Within this frame, when `tdlnorm()` is called with `L` and `U` set to their default values it behaves as `stats::dlnorm()`

```
tdlnorm(x, meanlog = 2, sdlog = 1)
```

but when called with different settings for `L` and `U`; such as:

`tdlnorm(x, meanlog = 2, sdlog = 1, L = 5, U = 20)`

it behaves as a lognormal density left truncated at `L=5` and right truncated at `U=20`.

This goal could be achieved by writing a `tdlnorm()` as:

```
tdlnorm <-  function (x, meanlog = 0, sdlog = 1,  L = 0,  H = Inf)
 {

  density <-
     stats::dlnorm(x, meanlog=meanlog, sdlog=sdlog)/
        (
        stats::plnorm(H, meanlog=meanlog, sdlog=sdlog)-
        stats::plnorm(L, meanlog=meanlog, sdlog=sdlog)
          )

   return(density)
 }
```

That returns the same results as function `truncdist::dtrunc()`

```
tdlnorm(x, 1, 2, L= 5, H = 20)
```

```
## [1] 0.11524 0.07297 0.05109
```

```
dtrunc(x, spec = "lnorm", a = 5, b = 20, meanlog = 1, sdlog = 2)
```

```
## [1] 0.11524 0.07297 0.05109
```

As this function clearly works, next step could be to write something similar for other distributions such as `weibull`, `gumbel` or `gamma`. We have to admit that all of this may become as quite time consuming.

A different approach could be to define a different function, `dtruncate()`, taking the name of a density distribution as an argument and returning a second function that computes density values for the truncated distribution:

```
dtruncate <-
  function (dist, pkg = stats){

    dist <- deparse(substitute(dist))
    envir <- as.environment(paste("package", as.character(substitute(pkg)), sep = ":"))

    ddist=paste("d", dist, sep = "")
    pdist=paste("p", dist, sep = "")

    #gets density function
    ddist <-  get(ddist, mode = "function", envir = envir)
    #gets argument of density function
    dargs <- formals(ddist)

    #gets probability function
    pdist <- get(pdist, mode = "function", envir = envir)
    #gets argument of probability function
    pargs <- formals(pdist)

    #Output function starts here
    density <- function ()
    {
      #check L U
      if (L > U) stop("U must be greater than or equal to L")

      #gets density arguments
      call <- as.list(match.call())[-1]

      #all unique arguments belonging to density and ddist
      dargs <- c(dargs[!is.element(names(dargs), names(call))], call[is.element(names(call), n

      #all unique arguments belonging to probability and pdist
      pargs <- c(pargs[!is.element(names(pargs), names(call))], call[is.element(names(call), n

      #select x only where defined by L and U
      dargs$x <- x[x > L & x <= U]

      #define arguments for pdist in L and U
      pUargs <-  pLargs <- pargs
      pUargs$q <- U
      pLargs$q <- L

      #initialize output
      density <- numeric(length(x))

      #standard method for computing density values for truncated distributions
      density[x > L & x <= U] <-  do.call("ddist", as.list(dargs)) / (do.call("pdist", as.list(pUarg

      #returns density values for truncated distributions
      return(density)

    }

    #add to density function formals L and U with values as passed with dtruncate
    formals(density) <-  c(formals(ddist), eval(substitute(alist(L=-Inf, U=Inf))))
    #return density function
    return(density)
  }
```

with:

- envir: the environment `plnorm()` and `dlnorm()` belong to

We can now define a new `tdlnorm()` as:

```
tdlnorm <- dtruncate(dist = lnorm)
```

and use it as:

```
p <- ppoints(1000)
x <- qlnorm(p, meanlog = 1, sdlog = 1)
d <- tdlnorm(x, meanlog = 1, sdlog = 1)
dt <- tdlnorm(x, meanlog = 1, sdlog = 1, L= 5, U = 10)
plot(x, dt, type = "n", xlab = "Quantile", ylab = "Density")
points(x, dt, type = "s", col = "red", lwd = 2)
points(x, d, type = "s", col = "darkblue", lwd = 2)
title("Truncated and not-truncated log-normal")
grid()
```

clearly, `tdlnorm()` returns the same results as `truncdist::dtrunc()`:

```
dtrunc(x = 5:8, spec = "lnorm", a = 5, b = 10, meanlog = 1, sdlog = 1)
```

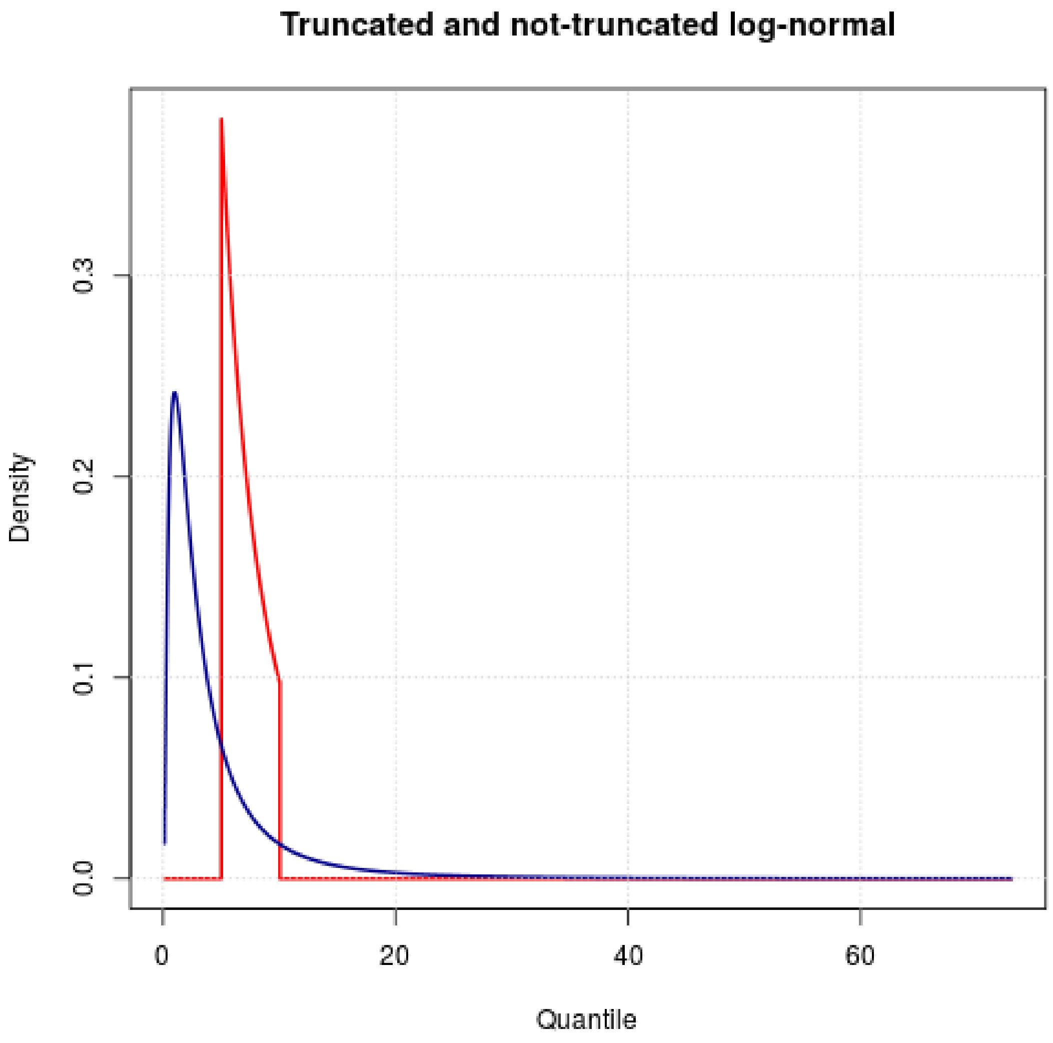


Figure 5.1: plot of chunk closures-030

```
## [1] 0.3792 0.2781 0.2085 0.1594
```

```
tdlnorm(x = 5:8, meanlog = 1, sdlog = 1, L = 5, U = 10)
```

```
## [1] 0.0000 0.2781 0.2085 0.1594
```

Moreover, our newly created `tdlnorm()` function takes as argument `meanlog` and `sdlog`, as well as `lower.tail = TRUE, log.p = FALSE, as stats::plnorm()` does, despite these arguments were not mentioned when calling `dtruncate()`.

Now that we have `dtruncate()`, the same exercise can be replicate, at no extra programming effort, to any density function:

```
dweibull <-  dtruncate(dist = weibull)
dgpd <- dtruncate(gpd, pkg = evd)
```

5.5 Functions with memory

When talking about clousures, we used the referencing environment of `f()` to hold any value passed by `g()`. Similarly, we can use the same environment to keep a state across multiple executions of `f()`.

5.5.1 Example: Track how many times a function is called

We may consider a function that simply returns the current date but tracks how many times it has ben called:

```
g <- function(){
 i <- 0
 f <- function(){
    i <<- i+1
    cat("this function has been called ", i, " times", "\n")
    date()
}}

f <- g()
#first call
f()
```

```
## this function has been called  1  times
```

```
## [1] "Mon Sep 22 16:33:25 2014"
```

```
#second call
f()
```

```
## this function has been called  2  times

## [1] "Mon Sep 22 16:33:25 2014"

#third call
f()

## this function has been called  3  times

## [1] "Mon Sep 22 16:33:25 2014"
```

Note that, we used the `<<-` operator that assigns in the parent environment. This is equivalent to:

```
assign("i", i+1, envir = parent.env(environment())):
```

5.5.2 Example: Avoid re-calculate previous results

We can use the referencing environment of a function to keep previous returned values of the same function. By using this idea, we could try to avoid re-calculating previously computed values.

Supose we want a function that takes `n` as argument and returns all primes less or equal to `n`. This function already exists within library `pracma`:

```
library(pracma)

## Error: there is no package called 'pracma'

primes(n = 9)

## Error: could not find function "primes"
```

In order to keep previous results we can define a function `makefprime()` that, when called, returns a second function with an environment `.env` attached:

```
makefprime = function () {
  .env = new.env()
  f = function(n) {
    symbol = paste("p", n, sep = ".")
    if (exists(symbol, envir = .env)){
      prime = get(symbol, envir = .env)
    }
    else {prime = primes(n = n)
      assign(symbol , prime, envir = .env)
    }
    prime
   }
f
}
```

We can now create a function named for instance `fprimes()` by calling function `makefprime()` which returns identical results when compared with `primes()`.

```
fprimes = makefprime()
fprimes(10)
```

```
## Error: could not find function "primes"
```

Now suppose we need to compute prime numbers several time within a working session or a for loop. When `n` is large, this computation may require a substancial ammount of time.

```
system.time({p1 = fprimes(n = 10^7)})
```

```
## Error: could not find function "primes"

## Timing stopped at: 0.001 0 0
```

Nevertheless, because of the way we defined `fprimes()`, second time this function is called with `n = 10^7` computing time is practicaly zero as the function reuse previously computed results as stored in environment `.env`.

```
system.time({p2 = fprimes(n = 10^7)})
```

```
## Error: could not find function "primes"

## Timing stopped at: 0 0 0.001
```

5.5.3 Example: Add to an existing plot

As a last example, we may want to have a function that add to an existing plot any time a new observation becomes available, using the same mechanism, we can define a `new_plot()` function that instances a new plot the first time it is called:

```
new_plot = function(){
  xx = NULL
  yy = NULL
  function(x, y, ...) {
  xx <<- c(xx, x)
  yy <<- c(yy, y)
  plot(xx, yy, ...)
}}

this_plot <- new_plot()
```

and add to the same plot at each next call:

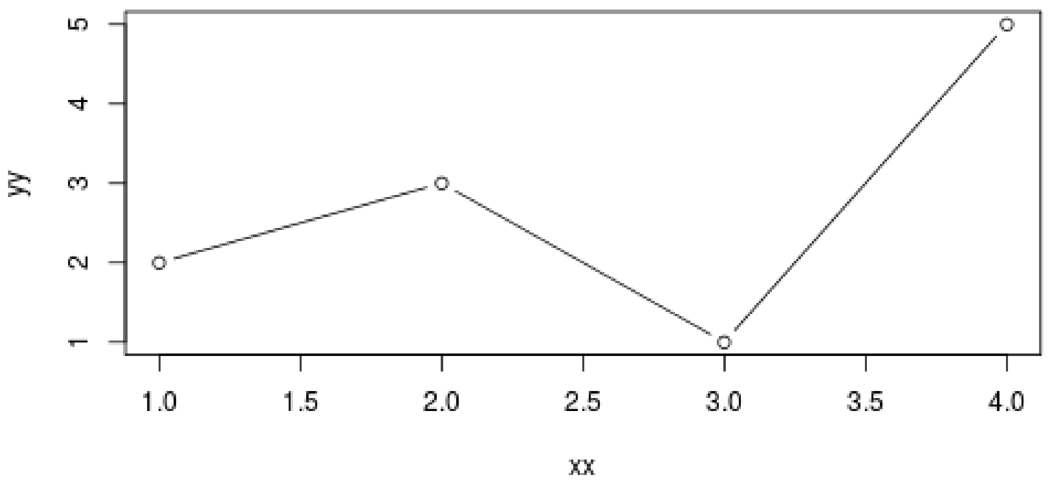


Figure 5.2: first call

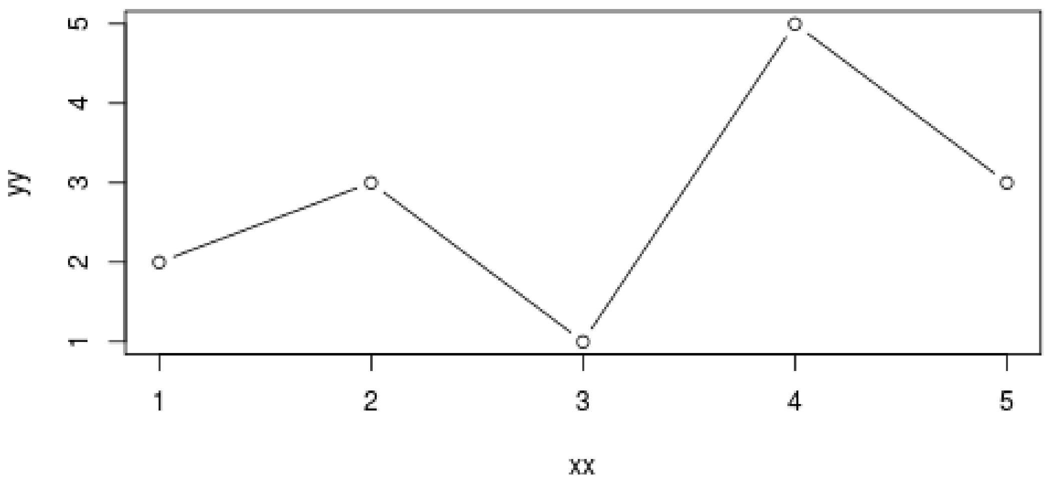


Figure 5.3: second call

```
this_plot (1:4, c(2, 3, 1, 5), type = "b")
```

```
this_plot(5, 3, type = "b")
```

```
this_plot(6, 3, type = "b", col = "red")
```

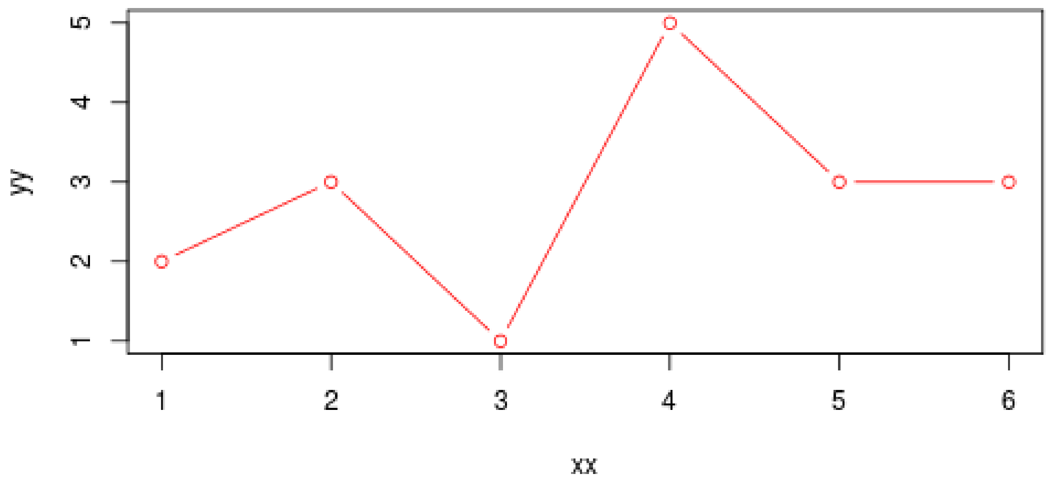


Figure 5.4: third call

5.6 Functionals

Functionals are functions that take a function as input and return a data object as output.

R incorporates many examples of functionals. Among many, `Reduce()` and `Filter()` are two good cases in point.

`Reduce(f, x)` tries to fold the element of `x` according to function `f()`. As a result we may use this function for binding the elements of a list into a matrix:

```
l <- list(x = 1:4, y = 4:1)
Reduce(rbind, l)
```

```
##      [,1] [,2] [,3] [,4]
## init    1    2    3    4
##         4    3    2    1
```

`Filter(f, x)` applies function `f()` to each element of `x`, and returns the subset of x for which this gives `TRUE`. In order to subset even number from any vector `x` we could write

```
Filter(f = function(x) x %% 2 == 0 , x = 1:5)
```

```
## [1] 2 4
```

As a very interesting example of functional we may define:

```
fun <- function(f, ...) f(...)
```

That is a function `fun()` that takes any function `f()` as input along with any other argument and compute `f(...)` where `...` represents the set of arguments.

As a result we may write:

```
fun(mean, x = 1:10, trim = .1)
```

```
## [1] 5.5
```

that is equivalent to:

```
mean(x = 1:10, trim = .1)
```

```
## [1] 5.5
```

Functionals are very often excellent substitutes to for loops as they allow to communicate the objective of our code in a more clear and concise manner as the code will be cleaner and it will more closely adhere to `R`'s idioms.

Functionals may even perform a little better than the equivalent for loop nevertheless but, in a first instance, our focus must always be on clarity rather than performances.

`lapply()` is, possibly, the most used functional:

```
lapply(list(one = 1, a = "a"), FUN = is.numeric)
```

```
## $one
## [1] TRUE
##
## $a
## [1] FALSE
```

`lapply()` can be consider as the main functional. `sapply()` and `vapply()` perform as `lapply()` but return a simplified output. `mapply()` and `Map()` are extension of `lapply()` that allow for multiple inputs.

5.6.1 lapply

`lapply()` takes a function and applies it to each element of a list, saving the results back into a result list. `lapply()` is the building block for many other functionals. In principle, `lapply()` is a wrapper around a standard for loop. The wrapper is written in `C` to increase performance.

`lapply()` takes three arguments:

- a list `X`, or anything that can be coerced to a list by `as.list()`
- a function `FUN` that takes, as first argument, each element of `X`
- the `''...''` argument that can be any argument to be passed to `FUN`

Suppose we want to gain the maximum of each column for the `airquality` data frame. By using a for loop we could write:

```
n <- ncol(airquality)
out <- numeric(n)
for (i in 1:n){
 out[i] <- max(airquality[,i], na.rm = TRUE)
}
out
```

```
## [1] 168.0 334.0  20.7  97.0   9.0  31.0
```

alternatively, as a data frame is a list:

```
lapply(X=airquality, FUN = max, na.rm = TRUE)
```

```
## $Ozone
## [1] 168
##
## $Solar.R
## [1] 334
##
## $Wind
## [1] 20.7
##
## $Temp
## [1] 97
##
## $Month
## [1] 9
##
## $Day
## [1] 31
```

The second chunk of code is by far more clear and concise than the first one even though a vector would be preferable than a list as output.

Moreover, `lapply()` as opposite to `for` loops does not produce any intermediate result when running. In the above `for` loop, the value of the result of the loop, vector `out`, changes at each iteration. The result of `lapply()`, instead, can be assigned to a variable but does not produce any intermediate result.

By default `lapply()` takes each element of list `X` as the first argument of function `FUN`. This works perfectly, as long as each element of `X` is the first of `FUN`. This is true in case we want to compute the mean of each column of a data frame as each column is passed as first argument to function `mean()`.

But, suppose we want to compute various trimmed means of the same vector, `trim` is the second parameter of `mean()`, so we want to vary `trim`, keeping the first argument `x` fixed.

This can be easily achieved by observing that the following two calls are equivalent:

```
mean(1:100, trim = 0.1)

## [1] 50.5

mean(0.1, x = 1:100)

## [1] 50.5
```

This is possible because `R` first matches formals by name and afterword by position.

As a result, in order to use `lapply()` with the second argument of function `FUN`, we just need to name the first argument of `FUN` and pass it to `lapply()` as part as the `''...''` argument:

```
x <- rnorm(100)
lapply(X = c(0.1, 0.2, 0.5), mean, x = x)

## [[1]]
## [1] 0.06386
##
## [[2]]
## [1] 0.07919
##
## [[3]]
## [1] 0.02135
```

5.6.2 sapply and vapply,

`sapply()` and `vapply()`, variants of `lapply()` that produce vectors, matrices and arrays as output, instead of lists.

`lapply()` returns a list as output, in order to get a vector, the previous examples could be written by using `sapply()`, a simple variant of `lapply()`, that tries to return an atomic vector instead of a list.

```
sapply(X=airquality, FUN = max, na.rm = TRUE)
```

```
##   Ozone Solar.R    Wind    Temp   Month     Day
##   168.0   334.0    20.7    97.0     9.0    31.0
```

Note that `sapply()` is a simple wrapper around `lapply()` that uses `simplify2array()`.

Unfortunately, `simplify2array()` and therefore `sapply()` offer very little control on the type of output that is returned:

```
sapply(list(), is.numeric)
```

```
## list()
```

In this case we would expect `sapply()` to return a empty logical vector, not an empty list.

As a result, `sapply()` may represent a excellent shortcut when working with R in interactive mode but not a good function to be used when developing serious R code.

A better alternative to `sapply()` is provided by `vapply()` a second variant of `lapply()` that allows to specify, by mean of argument `FUN.VALUE`, the kind of output we want the functional to return. In fact,

```
vapply(list(), is.numeric, FUN.VALUE = logical(1))
```

```
## logical(0)
```

```
vapply(X=airquality, FUN = max, na.rm = TRUE, FUN.VALUE = numeric(1))
```

```
##   Ozone Solar.R    Wind    Temp   Month     Day
##   168.0   334.0    20.7    97.0     9.0    31.0
```

In this case the `FUN.VALUE` argument specify what kind of output each element of the result should be. `vapply()` improves consistency by providing either the return type we were expecting or error. This is a clear advantage, as it helps catch errors before they happen and leads to more robust code.

As an example, suppose we have a list of data frames:

```
df_list <- list(cars, airquality, trees)
```

and that we need to know the number of columns of each data frame returned in a vector. We can easily achieve this goal by:

```
sapply(df_list, ncol)
```

```
## [1] 2 6 3
```

Nevertheless, suppose we want to apply the same function to a very large list of data frame and, by chance, one of them happens to be `NULL`.

```
df_list <- list(df1 = cars, df2 = NULL, df3 = trees)
```

When using `sapply()` a `list`, instead of a `vector` is returned

```
sapply(df_list, ncol)
```

```
## $df1
## [1] 2
##
## $df2
## NULL
##
## $df3
## [1] 3
```

If we use `vapply()` instead:

```
vapply(df_list, ncol, FUN.VALUE = numeric(1))
```

```
## Error: values must be length 1,
##  but FUN(X[[2]]) result is length 0
```

`R` returns an error.

Clearly this second behavior is more coherent and, within the frame of a large project, possibly helps to avoid annoying hours of debugging

Finally, `vapply()` is faster that `sapply()` as `R` does not have to *guess* the kind of output `sapply()` needs to return.

Suppose we have a list made of `10^6` vectors of variable length between one and five.

```
n <- sample(1:5, 10^6 , rep = T)
vector_list <- lapply(n, sample , x = 0:9)
```

We can appreciate the difference in speed between `sapply()` and `vapply()` by the following example:

```
system.time(
  sapply(vector_list, length )
)
```

```
##    user  system elapsed
##   3.454   0.016   3.470
```

```
system.time(
  vapply(vector_list, length, FUN.VALUE = numeric(1))
)
```

```
##    user  system elapsed
##   0.382   0.008   0.390
```

5.6.3 lapply patterns

When using `lapply()` we can loop at least in two different ways: on the `xs` or on an index `i`. As a example, we may consider the following code:

```
x = 1:3
vapply(x, function(x) x*x, numeric(1))
```

```
## [1] 1 4 9
```

and compare it with the next chunk:

```
vapply(1:length(x), function(i , x) x[i]*x[i], x=x , FUN.VALUE=numeric(1))
```

```
## [1] 1 4 9
```

The second chunk of code looks a little more complicated as it introduces the `i` index that in this case is simply redundant.

Suppose, instead, we want to compute the mean the three variables of the `trees` dataset but using different trims values for all columns and setting `na.rm = TRUE`.

By using a for loop, beside clarity and the number of lines required, the code is quite simple:

```
n = ncol(trees)
out = numeric(n)
trim = c(0.1, 0.2, 0.3)
for ( i in 1:n){
  out[i] = mean(trees[,i], trim[i], na.rm = TRUE)
}
out
```

```
## [1] 13.14 76.58 25.43
```

When translating this code with `lapply()`, we may use the index strategy result as mandatory because, `lapply(X, FUN, ...)` allows only argument `X` of function `FUN` to vary. All other arguments to function `FUN` can be passed via ‘‘`...`” but without varying.

```
lapply(1:ncol(trees), function(i, x, trim, ...) mean(x[, i], trim[i], na.rm = TRUE),
    x = trees, trim = c(0.1, 0.2, 0.3))
```

```
## [[1]]
## [1] 13.14
##
## [[2]]
## [1] 76.58
##
## [[3]]
## [1] 25.43
```

The same approach we used to solve `for()` loops into `lapply()`, can be generalized to nested loop.

As an example consider this apparently messy loop:

```
ni = 4
nj = 2

nk = ni*nj
k = numeric(nk)
for (j in 1:nj){
  if ( j %% 2 == 0){
    for ( i in 1:ni){
      if ( i %% 2 == 0 ) next_k = i+j
      else next_k = i-j
      cat("first ij " , i , j , next_k, "\n")
      k[i+(j-1)*ni] = next_k
    }
  }
  else {
    for (i in 1:ni){
    next_k <- 99
    cat("second ij " , i , j , next_k, "\n")
    k[i+(j-1)*ni] = next_k
    }
  }
}
```

```
## second ij  1 1 99
## second ij  2 1 99
## second ij  3 1 99
## second ij  4 1 99
## first ij  1 2 -1
## first ij  2 2 4
## first ij  3 2 1
## first ij  4 2 6
```

We have at least two alternatives to transform this loop.

We can use `lapply()` with the index strategy. We have to rewrite the innermost part of the whole loop as:

```
f = function(k , i , j) {
  i <- i[k]
  j <- j[k]
  result <- 99
  if( j %% 2 == 0){
  result <- ifelse(i %% 2 == 0 , i+j , i-j)
  }
  result
}
```

secondly, with a little help from `expand.grid`:

```
grid <- expand.grid(i = 1:ni, j = 1:nj )
with(grid , vapply(1:nrow(grid), f, i=i , j=j, FUN.VALUE = numeric(1)))
```

```
## [1] 99 99 99 99 -1  4  1  6
```

The use of `expand.grid()` allows to transform a nested loop into a matrix of all possible combinations over which we can easily loop by using `lapply()`. This approach may help a lot in simplifying our code but, is quite memory hungry as it requires to explode all possible combinations in a single matrix.

Alternatively, we can make use of a nested `lapply()`. We first define function `f()` as:

```
f = function(i , j) {
  ifelse( j %% 2 == 0,
    ifelse(i %% 2 == 0 , i+j , i-j),
  99)
}
```

Afterword, we nest two functionals as following:

```
unlist(lapply(1:2, function(j, i = ni) vapply(1:4, FUN = f, j, FUN.VALUE = numeric(1))))
```

```
## [1] 99 99 99 99 -1  4  1  6
```

This case clearly illustrate how much we gain in clarity and efficiency when using functionals instead of loops.

As an alternative to `lapply()` with the index strategy we may consider `mapply()` and `Map()` that naturally iterate over multiple input data structures in parallel.

5.6.4 mapply and Map

A first alternative to `lapply()`, along with the index strategy, is represented by `mapply()`:

```
mapply(mean ,trees, trim = c(0.1 , 0.2, 0.3),
  MoreArgs = list(na.rm = TRUE),
  SIMPLIFY = FALSE)
```

```
## $Girth
## [1] 13.14
##
## $Height
## [1] 76.58
##
## $Volume
## [1] 25.43
```

The structure of `mapply()` is:

```
mapply(FUN, ..., MoreArgs=NULL, SIMPLIFY = TRUE)
```

where, as opposite to `lapply()`, the `FUN` argument takes the first position and the ‘’`...`” argument specifies any list of arguments to be passed to `FUN` during the iteration.

The `MoreArgs` argument takes a list of parameters to be kept as fixed during the iteration. Note that this breaks R’s usual lazy evaluation semantics, and is inconsistent with other functions.

The `SIMPLIFY` as set to `TRUE` by default, allows output simplification in the `sapply()` fashion. Clearly, this options gives as little control over the output as `sapply()` does.

An alternative to `mapply()` is represented by `Map()` that returns identical results to `mapply()` with `SIMPLIFY` set to `FALSE` and uses anonymous or external function to pass fixed parameters to `FUN`.

```
Map(function(...) mean(..., na.rm = TRUE),
  x = trees , trim = c(0.1 , 0.2, 0.3))
```

```
## $Girth
## [1] 13.14
##
## $Height
## [1] 76.58
##
## $Volume
## [1] 25.43
```

The choice between using `mapply()` or `Map()` is surely a personal one.

Both `Map()` and `mapply()` can be used to substitute nested loops.

We can consider the previous nested loop and, with a little help from `expand.grid()` and function `f()`

```
f = function(i , j) {
  result = 99
  if( j %% 2 == 0){
  result <- ifelse(i %% 2 == 0 , i+j , i-j)
  }
  result
}

grid <- expand.grid(i = 1:4, j = 1:2)
```

And, similarly to the `lappy()` case, we could write:

```
with(grid , mapply(f, i=i , j=j, SIMPLIFY = TRUE))

## [1] 99 99 99 99 -1  4  1  6
```

or:

```
unlist(with(grid , Map(f, i=i , j=j)))

## [1] 99 99 99 99 -1  4  1  6
```

5.6.5 eapply

Sometimes we may find our self using environments as data structure because of many reasons including: hash tables and copy on modify semantic that does not apply to environments.

When objects are stored within an environment, we can use `eapply()` as a functional to the environment:

```
env <- new.env()
env$x = 3 ; env$y = -2
eapply(env, function(x) ifelse(x>0 , 1 , -1))

## $x
## [1] 1
##
## $y
## [1] -1
```

5.7 Anonymous functions

In R, we usually assign functions to variable names. Nevertheless, functions can exists without been assigned to symbol. Functions that don't have a name are called **anonymous functions**.

We can call anonymous functions directly, as we do with named functions, but the code is a little unusual as we have to use brackets both to include the whole function definition and to pass arguments to the function:

```
(function(x) x + 3)(10)
```

```
## [1] 13
```

Note that this is exactly the same as:

```
f <- function(x) x + 3
f(10)
```

```
## [1] 13
```

We use anonymous functions when it's not worth the effort of assigning functions to a name. We could plot a function `s(x)` by:

```
s <- function(x) sin(x)/sqrt(x)
integrate(s,  0, 4)
```

```
## 1.61 with absolute error < 4.5e-11
```

or alternatively by:

```
integrate(function(x) sin(x)/sqrt(x),  0, 4)
```

```
## 1.61 with absolute error < 4.5e-11
```

in this case `function(x) sin(x)/sqrt(x)` is an example of anonymous function.

Finally, anonymous functions are, by all rights, normal R functions as they have `formals()`, a `body()`, and a parent `environment()`:

```
formals(function(x) x+1)
```

```
## $x
```

```
body(function(x) x+1)
```

```
## x + 1
```

```
environment(function(x) x+1)
```

```
## <environment: R_GlobalEnv>
```

5.8 Lists of functions

Functions, as any type of `R` object, can be stored in a list.

```
fun_list <- list(m = mean , s = sd)
```

This makes it easier to work with groups of related functions.

Functions defined within a list are still accessible at least in three different ways:

using function `with()`

```
with (fun_list, m(x = 1:10))
```

```
## [1] 5.5
```

by using the `$` operator

```
fun_list$m( x = 1:10)
```

```
## [1] 5.5
```

by attaching the list:

```
attach(fun_list)
m( x = 1:10)
```

```
## [1] 5.5
```

```
detach(fun_list)
```

Lists of functions can be most useful when we want to apply all functions of the list to the same set of data.

We can achieve this goal in two logical steps.

We first define a function

```
fun <- function(f, ...){f(...)}
```

that takes a function `f()` as argument along with any other arguments '`...`" and returns `f(...)`. In practice:

```
fun(mean, x = 1:10, na.rm = TRUE)
```

```
## [1] 5.5
```

Secondly, we apply function `fun()` to the list of functions. Arguments required by the functions stored in the list are passed by the "`...`" argument:

```
lapply(fun_list, fun, x = 1:10)
```

```
## $m
## [1] 5.5
##
## $s
## [1] 3.028
```

Under almost all circumstances, equivalent results can be achieved by using function `do.call()` within a call to `lapply()`:

```
lapply(fun_list, do.call, list(x = 1:10, na.rm = T))
```

```
## $m
## [1] 5.5
##
## $s
## [1] 3.028
```

the only difference being that arguments to functions within the list must be enclosed in a list too.

5.8.1 Example: Multiple *Anderson-Darling* tests

As a simple example we may want to compare the results of four **Anderson-Darling** type tests from the `truncgof` package applied to the same data.

We can define a list that holds these four functions and store it in the global environment:

```
require(truncgof, quietly = TRUE)
```

```
##
## Attaching package: 'truncgof'
##
## The following object is masked from 'package:stats':
##
##     ks.test
```

```
nor_test <- list(ad2.test = ad2.test, ad2up.test = ad2up.test, ad.test = ad.test,
    adup.test = adup.test)
```

and, afterword, apply function `fun()` to each element of this list:

```
x <- rnorm(100, 10, 1)
m <-  mean(x)
s <- sd(x)
lapply(nor_test, fun, x , distn = "pnorm", list(mean = m, sd = s), sim = 100)
```

```
## $ad2.test
##
##  Quadratic Class Anderson-Darling Test
##
## data:  x
## AD2 = 0.3352, p-value = 0.55
##
## treshold = -Inf, simulations: 100
##
##
## $ad2up.test
##
##  Quadratic Class Anderson-Darling Upper Tail Test
##
## data:  x
## AD2up = 1.527, p-value = 0.93
##
## treshold = -Inf, simulations: 100
##
##
## $ad.test
##
##  Supremum Class Anderson-Darling Test
##
## data:  x
## AD = 2.807, p-value = 0.24
## alternative hypothesis: two.sided
##
## treshold = -Inf, simulations: 100
##
##
## $adup.test
##
##  Anderson-Darling Upper Tail Test
##
## data:  x
## ADup = 10.71, p-value = 0.54
## alternative hypothesis: two.sided
##
## treshold = -Inf, simulations: 100
```

5.8.2 Example: Summary statistics

We may want to define a function that returns some specific statistics for a given set of variables in the form of a `data.frame`.

```
this_summary <- as.data.frame(rbind(vapply(trees, mean, FUN.VALUE = numeric(1)),
    vapply(trees, sd, FUN.VALUE = numeric(1)), vapply(trees, function(x, ...) {
        diff(range(x))
    }, FUN.VALUE = numeric(1))))

row.names(this_summary) <- c("mean", "sd", "range")
this_summary
```

```
##        Girth Height Volume
## mean  13.248 76.000  30.17
## sd     3.138  6.372  16.44
## range 12.300 24.000  66.80
```

We may achieve the same result by writing a more general function that will work with any kind of statistics as long as they return a single value:

```
my_summary <- function(x, flist) {
    f <- function(f, ...) f(...)
    g <- function(x, flist) {
        vapply(flist, f, x, FUN.VALUE = numeric(1))
    }
    df <- as.data.frame(lapply(x, g, flist))
    row.names(df) <- names(flist)
    df
}

my_summary(cars, flist = list(mean = mean, stdev = sd, cv = function(x, ...) {
    sd(x, ...)/mean(x, ...)
}))
```

```
##         speed    dist
## mean  15.4000 42.9800
## stdev  5.2876 25.7694
## cv     0.3434  0.5996
```

5.8.3 fapply

Working with this mind set we may even define a function `fapply()` that applies all functions of a list to the same set of arguments

```
fapply <- function(X, FUN, ...){
  lapply(FUN, function(f, ...){f(...)}, X, ...)
}
```

and use it as:

```
basic_stat <- list(mean = mean, median = median, sd = sd)
fapply(1:10, basic_stat)
```

```
## $mean
## [1] 5.5
##
## $median
## [1] 5.5
##
## $sd
## [1] 3.028
```

Chapter 6

Creating R packages

6.1 Introduction

A wide documentation is available on the internet providing the technical instructions to build `R` packages, starting from the manual Writing R extensions[1] at the official `R` site.

This section provides a general overview about R packages. The following sections try to provide some insights on the mechanism used by `R` when dealing with packages.

6.1.1 Why create a `R` package?

According to Rossi[2], there are at least three good reasons to create an R package.

1. Creating an R package forces the user to document the code and provide test examples to insure that it actually works. It will also be much easier to use the code as documentation will only be a `?` command away and all of functions and shared libraries will be available for use. This is a good reason to create a package, also for its own use.

2. If the goal is disseminate a research, this is an ideal way of making sure others have access to the work. It will also increase the probability that eventually the work will be correct. This is a good reason to create a package for a team (private) use.

3. Giving back something to this amazing community of volunteers! This is a good reason to make the package available for the whole world (e.g. through CRAN).

6.1.2 Structure of a package

The sources of a R package consists of a subdirectory containing some files and directories in a well organized structure.

[1] http://cran.r-project.org/doc/manuals/r-release/R-exts.html
[2] http://www.math.ncu.edu.tw/~chenwc/R_note/reference/package/packages.pdf

Files 'DESCRIPTION' and 'NAMESPACE' and subdirectories 'R', 'man', and 'data' are required for every package. The base `package.skeleton()` function creates all the required directories.

The 'DESCRIPTION' file contains basic information about the package. The 'Package', 'Version', 'License', 'Description', 'Title', 'Author', and 'Maintainer' fields are mandatory, all other fields are optional.

R has a namespace management system for code in packages. This system allows the package writer to specify which variables in the package should be exported to make them available to package users, and which variables should be imported from other packages. The mechanism for specifying a namespace for a package is through the 'NAMESPACE' file in the top level package directory.

The 'R' subdirectory contains R code files, only. The `package.skeleton()` function returns a .R file for each function.

The 'man' subdirectory should contains only documentation files for the objects in the package in R documentation (Rd) format.

The 'data' subdirectory contains data files in R format.

A package may also contain files 'INDEX', 'configure', 'cleanup', 'LICENSE', 'LICENCE', 'COPYING' and 'NEWS' and directories 'data', 'exec', 'inst', 'po', 'src', and 'tests'. These subdirectories can be missing, but which should not be empty.

The sources and headers for the compiled code are in 'src'. The 'demo' subdirectory is for R scripts (for running via `demo()`) that demonstrate some of the functionality of the package. The contents of the 'inst' subdirectory will be copied recursively to the installation directory. Subdirectories of 'inst' should not interfere with those used by R. Subdirectory 'exec' could contain additional executables the package needs, typically scripts for interpreters such as the shell, Perl, or Tcl. This mechanism is currently used only by a very few packages, and still experimental. Subdirectory 'tests' is for additional package-specific test code, similar to the specific tests that come with the R distribution. Subdirectory 'po' is used for files related to localization.

6.1.3 Creating a package

When all is well organized, the following steps are required in order to create a package:

1. the 'DESCRIPTION' file ought be filled in with required information,
2. the 'NAMESPACE' file ought be filled in with required information,
3. the R documentation files ought be written,
4. the sources and headers for the compiled code, if any, ought be contained in the 'src' directory.

Then, a package can be created. This requires three steps:

1. **build**: the shell command `R CMD BUILD` builds an R source tarball. This means that temporary files are removed from the source tree of the package and everything is packed into a single file.

2. **check**: the shell command `R CMD CHECK` runs a wide variety of diagnostic checks on the package. Checks may be run before or after the build step.
3. **install**: the shell command `R CMD INSTALL` installs the package into a library and makes it available for usage in R. The R function `install.packages()` can be used instead.

R package `devtools` provide a lot of useful function in order to help developers to develop their own package. Moreover, RStudio IDE integrates `devtools` providing an user interface to build packages.

6.1.4 Writing R documentation files

R objects are documented in files written in "R documentation" (Rd) format, a simple markup language much of which closely resembles Latex, which can be processed into a variety of formats, including Latex, HTML and plain text.

An 'Rd' file consists of three parts. The header gives basic information about the name of the file, the topics documented, a title, a short textual description and R usage information for the objects documented. The body gives further information; for example, on the function's arguments and return value, as in the example. Finally, there is an optional footer with keyword information. The header is mandatory. Information is given within a series of sections with standard names (and user-defined sections are also allowed). Unless otherwise specified these should occur only once in an 'Rd' file (in any order).

The `roxygen2` R package allows to get a in-source documentation. Accordint to package vignettes[3], Roxygen2 provides a number of advantages over writing .Rd files by hand:

- Code and documentation are adjacent so when you modify your code, it's easy to remember that you need to update the documentation.
- Roxygen2 dynamically inspects the objects that it's documenting, so it can automatically add data that you'd otherwise have to write by hand.
- It abstracts over the differences in documenting S3 and S4 methods, generics and classes so you need to learn fewer details.

6.2 The package structure behind `R`

Packages provide a mechanism for loading optional code, data and documentation as needed.

An R package can be thought of as the software equivalent of a scientific article: articles are the *de facto* standard to communicate scientific results, and readers expect them to be in a certain format. R packages are a comfortable way to maintain collections of R functions and data sets (Leisch, 2009)[4].

The R distribution itself includes about 30 packages. With regard to the "importance" of a package, packages can be split into three categories.

- **Base** packages: part of the R source tree, maintained by R Core.

[3] http://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html
[4] http://cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf

- **Recommended** packages: part of every R installation, but not necessarily maintained by R Core.
- **Contributed** packages: all the rest. This does not mean that these packages are necessarily of lesser quality than the above, e.g., many contributed packages on CRAN are written and maintained by R Core members. The goal is simply to try to keep the base distribution as lean as possible.

The `installed.packages()` function returns a matrix with several information about installed packages. The "Priority" column reports the category (base, reccomended or contributed) which each package belong to.

Terms about R packages are often confused. This may help to clarify:

- **Package**: a collection of R functions, data, and compiled code in a well-defined format.
- **Library**: the directory where packages are installed.
- **Repository**: A website providing packages for installation.
- **Source**: The original version of a package with human-readable text and code.
- **Binary**: A compiled version of a package with computer-readable text and code, may work only on a specific platform.

6.3 Packages Environments

Every R package has three associated environments:

1. package environment
2. namespace environment
3. imports environment

The package environment contains all functions of the package exposed to the end user.

The namespace environment contains all functions the package including those functions included in the package environment. This is not a duplication of the functions contained in both environments as two equal functions in the two environment share the same memory address.

As a simple proof of concept, consider a first environment `env1` with a simple function `f()` in it:

```
env1 <- new.env()
env1$f <- function() NULL
```

and a second environment `env2` with an other function `f()` that is a copy of function `f()` from `env1`:

```
env2 <- new.env()
env2$f <- env1$f
```

with the help of function `mem_add()`, we can see that `env1$f` and `env2$f` share the same memory address:

```
mem_add <- function(x) substring(capture.output(.Internal(inspect(x))), 2, 17)
identical(mem_add(env1$f) , mem_add(env2$f))
```

```
## [1] TRUE
```

Similarly, with the help of function `getAnywhere()` we can see that function `mean()` is located both in the package environment and in the namespace environment of package `base`.

```
getAnywhere(mean)$where
```

```
## [1] "package:base"   "namespace:base"
```

Having functions within namespaces rather than packages allows the developer to expose to the end user only those functions that are supposed to be called directly and hide all those functions that are to be internally called from exposed function.

As a common practice, namespace environments may hold a quite large number of functions. As an example we can consider package `stats`; this package contains 452 objects exposed to the end user while the corresponding namespace has 1114 objects:

```
length(as.environment(.getNamespace("stats")))
```

```
## [1] 1114
```

```
length(as.environment("package:stats"))
```

```
## [1] 452
```

The imports environment of a package contains objects from other packages that are explicitly stated requirements for a package to work properly. Most packages published on CRAN are not islands; they build on functionality provided in other packages.

We can get the names of the packages any package requires by using a little variant of function `packageDescription()`:

```
package_imports <- function(pkg){
  v <- packageDescription(pkg, fields = "Imports")
  d <- data.frame( strsplit(v, split = ",")[[1]])
  names(d) <- paste ("imports", pkg, sep = "_")
  d
}
```

and test it on package `ggplot2`

```
package_imports(pkg = "ggplot2")
```

```
##       imports_ggplot2
## 1     plyr (>= 1.7.1)
## 2              digest
## 3                grid
## 4   gtable (>= 0.1.1)
## 5            reshape2
## 6 \nscales (>= 0.2.3)
## 7               proto
## 8                MASS
```

We can also count the functions within the imports of `ggplot2`

```
vapply(getNamespaceImports("ggplot2"), length, FUN.VALUE = numeric(1))
```

```
##     base   digest     grid   gtable     plyr    proto reshape2   scales
##        1        3      198       18       74       13        6       73
##     MASS     MASS  methods
##        1        1        1
```

Notice that, `getNamespaceImports()` also shows an object from the `base` package. This object is not a function but a simple logical set to `TRUE`.

6.4 Calling a function from a package

As the package environment is included in the search path, when we call a function from a package, `R` looks for that function along the search path until it finds it in the package environment. Nevertheless, if we ask to a function from a package environment for the environment it belongs to, we have a little surprise:

```
environment( sd)
```

```
## <environment: namespace:stats>
```

That is, the environment of function `sd()` from `stats` package environment refers to the `stats` namespace environment as its environment.

As a consequence, when function `sd()` runs, a new environment is created whose enclosure is the `stats` namespace environment and all *hidden* functions within the namespace become available to function `sd()`.

The namespace environment of any package, as any environment has a parent. We can query `R` for the parent of any namespace:

```
parent.env(as.environment(.getNamespace("stats")))
```

```
## <environment: 0x3194ff0>
## attr(,"name")
## [1] "imports:stats"
```

As we can see the parent environment of a namespace is a new environment: `imports:packagename` whose parent, is the namespace of package `base`.

```
parent.env(parent.env(as.environment(.getNamespace("stats"))))
```

```
## <environment: namespace:base>
```

```
parent.env(parent.env(as.environment(.getNamespace("utils"))))
```

```
## <environment: namespace:base>
```

Finally, the parent of `namespace:base` happend to be our `R_GlobalEnv`.

```
parent.env(parent.env(parent.env(as.environment(.getNamespace("stats")))))
```

```
## <environment: R_GlobalEnv>
```

The following picture, borrowed from obeautifulcode.com[5] illustrates the whole chain of environments.

In practice, when we look for a function `f()` in a package `pkg`, we find `f()` in the package environment og `pkg`. When we call `f()`, it runs within the namespace of `pkg` and as this is its environment. The enclosure of execution environment of `f()` is therefore the namespace of `pkg`. Whenever `f()` calls a second function `g()`, `g()` is searched first in the execution environment and, as reasonably `g()` is not defined in there, `R` looks for `g()` in the name space and, in case `g()` does not belong to package `pkg`, `R` looks in the imports environment of `pkg`.

This search structure makes perfect sense as it increases the probaility of finding any function `g()` in the shortes possible time.

In case `g()` is not found in the imports of `pkg` then `R` looks for `g()` in the namespace of `base`. Again, this is very reasonable as, almost all packages have to refer to the base package.

If we assume that the dependency structure of the package has been built properly, the search should end up at the imports namespace or, at worse at the base namespace. In fact, if `g()` is not found within the namespace of `base` the next step in the search mechanism, points to our `R_GlobalEnv` and after that it moves down the search path until it either finds `g()` or reaches the empty environment.

Older version of `R` did not implement this idea of namespace and, as a consequence, the imports environment did not exists. Dependencies between packages was implemented with the use of *depends*. If a package `pkg1` was dependending from package `pkg2`, `pkg2` was attached in the search path just after package `pkg1`. Nowadays, few packages still use this idea of depends. As an exampel consider package `abc`: before attaching the package the search path should look like:

```
## [[1]]
## NULL
```

[5] http://obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/

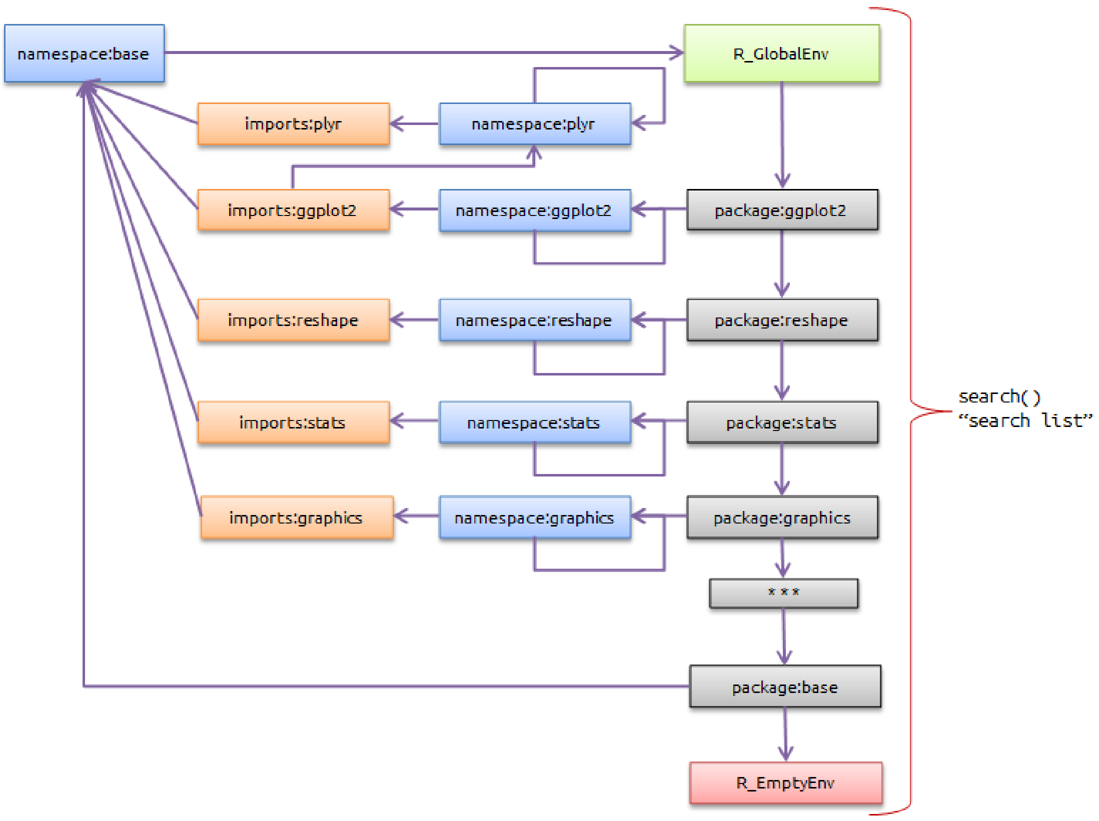


Figure 6.1: http://obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "Autoloads"         "package:base"
```

If we load package `abc version 1.8` and we run `search()` again, we observe how our search path is changed:

```
## Loading required package: SparseM
## Loading required package: methods
##
## Attaching package: 'SparseM'
##
## The following object is masked from 'package:base':
##
##     backsolve
##
## locfit 1.5-9.1    2013-03-22
```

```
##  [1] ".GlobalEnv"        "package:abc"       "package:locfit"
##  [4] "package:MASS"      "package:quantreg"  "package:SparseM"
##  [7] "package:methods"   "package:nnet"      "package:stats"
## [10] "package:graphics"  "package:grDevices" "package:utils"
## [13] "package:datasets"  "Autoloads"         "package:base"
```

Packages `nnet`, `quantreg`, `MASS` are attached as package `abc` depends on them while package `SparseM` is attached because package `quantreg` depends on it:

```
## [1] "R (>= 2.10), nnet, quantreg, MASS, locfit"
```

```
## [1] "R (>= 2.6), stats, SparseM"
```

Clearly, `imports` is to be preferred to `depends` as it offer a neter structure for `R` searching mechanism.

6.5 Loading packages

We load a package, with a call either to `library()` or `require()`. These functions perform very similarly but they have few very important differences that is worth to mention.

As a first difference we can notice the different messages the two functions requrn when called with a non existing library:

```
library("not_exist")
```

```
## Error: there is no package called 'not_exist'
```

```
require("not_exist")
```

```
## Loading required package: not_exist
```

```
## Warning: there is no package called 'not_exist'
```

beside the message, a more important side effect comes up when we assign the results of these calls to an object:

```
test_library <- library("not_exist")
```

```
## Error: there is no package called 'not_exist'
```

```
test_require <-  require("not_exist")
```

```
## Loading required package: not_exist
```

```
## Warning: there is no package called 'not_exist'
```

```
test_library
```

```
## Error: object 'test_library' not found
```

```
test_require
```

```
## [1] FALSE
```

We can notice that, in case of error, `library()` does not assign while `require()` returns `FALSE`. This behavior of `require()` allow us to make libraries loading more robust expecially within the body of a function:

```
if(!require("ggplot2")) {install.packages("ggplot2"); require("ggplot2")}
```

```
## Loading required package: ggplot2
```

Despite this difference, It's bad practice to use `library()` or `require()` inside a function, because it makes it hard to understand code dependencies. They should either be outside functions or, even better, in package `DESCRIPTION`.

When loading a package these four actions occur:

1. The namespace environment is loaded
2. A new environment is created: the package environment
3. Only *exported* functions are *copied* from the namespace to the package environment
4. Package environment is that included in the search list

Packages are usually loaded by mean of a *lazy loading* mechanism. Lazy loading is always used for code in packages but is optional, as it selected by the package maintainer, for datasets in packages. When a package namespace is loaded, the namespace environment is populated with *promises* for all the named objects and those objects specified in the `NAMESPACE` field of the package are copied into the package environment: when these promises are evaluated they load the actual code from a database.

There are separate databases for code and data, stored respectively in the `R` and `data` sub-directories. Each database consists of two files, `name.rdb` and `name.rdx`. The `.rdb` file is a concatenation of serialized objects, and the `.rdx` file contains an index. The objects are stored in a gzip-compressed format.

The loader for a lazy-load database of code or data is function `lazyLoad()` in the base package.

As an example we can write a function that load an existing object as a promise from a `.rdb` file being part of a `R` package without loading the package:

```
get_from_rdb <- function(symbol, filebase, envir =parent.frame()){
  lazyLoad(filebase = filebase, envir = envir, filter = function(x) x == symbol)
}
```

and use this function to get the promises in our local environment as:

```
Rlib = .libPaths()[1]
get_from_rdb(symbol =  "venice", filebase = file.path(Rlib, "evd/data/Rdata"))

## NULL

find("venice")

## [1] ".GlobalEnv"
```

Once we have the promise we can evaluate it in order to get the value associated to the promise:

```
head(eval(venice))

##        1   2   3   4   5  6  7  8  9 10
## 1931 103  99  98  96  94 89 86 85 84 79
## 1932  78  78  74  73  73 72 71 70 70 69
## 1933 121 113 106 105 102 89 89 88 86 85
## 1934 116 113  91  91  91 89 88 88 86 81
## 1935 115 107 105 101  93 91 NA NA NA NA
## 1936 147 106  93  90  87 87 87 84 82 81
```

Finally, we can easily load all data and functions from package `.rdb` files by:

```
lazyLoad(filebase = file.path(Rlib, "evd/data/Rdata"), envir = parent.frame(), filter = function(x) T

## NULL

lazyLoad(filebase = file.path(Rlib, "evd/R/evd"), envir = parent.frame(), filter = function(x) TRUE)

## NULL
```

Chapter 7

Object Oriented

7.1 Object Oriented Programming

Object oriented programming is a programming paradigm based on *classes* and *methods*.

A *class* is an abstract definition of a concrete real world object. A class is generally made of ordered and named slots.

For instance, a rectangle is defined given the lengths of its sides. Therefore objects of class rectangle can be defined by a class containing two slots of type numeric, named for instance `x` and `y`, corresponding to its sides. The specific rectangle of sides `x = 3` and `y = 6` represents an instance of the class rectangle.

Given a class, any number of dedicated methods can be written for that class. A specific *method*, for a given class, is a function that performs specific actions on an object of that class. Specific methods are defined as particular cases of general methods.

This mechanism is almost everywhere in `R`. For instance:

```
## Loading required package: methods
```

```
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

When calling `head(cars)`, `R` understand that `cars` is an object of class data.frame, `head` is a generic method and therefore, `R` looks for a specific head method for objects of class `data.frame`. This method exists and is named `head.data.frame` as shown by:

```
methods("head")
```

```
## [1] head.data.frame* head.default*    head.ftable*     head.function*
## [5] head.matrix      head.table*
##
##    Non-visible functions are asterisked
```

As `head.data.frame` is defined as a non-visible function within the namespace of `utils`, its content can be visualized by typing:

```
utils:::head.data.frame
```

```
## function (x, n = 6L, ...)
## {
##     stopifnot(length(n) == 1L)
##     n <- if (n < 0L)
##         max(nrow(x) + n, 0L)
##     else min(n, nrow(x))
##     x[seq_len(n), , drop = FALSE]
## }
## <bytecode: 0x31e1618>
## <environment: namespace:utils>
```

Finally, when calling method `head` on an object of class `function`:

```
head(lm)
```

```
##
## 1 function (formula, data, subset, weights, na.action, method = "qr",
## 2     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
## 3     contrasts = NULL, offset, ...)
## 4 {
## 5     ret.x <- x
## 6     ret.y <- y
```

According to the same mechanism, `R` returns the first six row of the `lm()` function by calling `utils:::head.function`.

End users are not interested in the class structure itself but do care about methods that are available to access the class. The `R` way of reaching this goal is to use generic functions and method dispatch: the same function performs different computations depending on the types of its arguments.

`R` is both interactive and has a system for object-orientation. The interactive component of `R` is a great tools for data analysis and quick development. Nevertheless, when it comes to software development, especially software development at enterprise level, a serious object oriented programming system is recommended.

R tries to achieve a compromise between object orientation and interactive programming and, although compromises are never optimal with respect to all goals they try to reach, they often work surprisingly well in practice.

Being able to understand when interactive programming has to be converted and structured into an object oriented library is a key point to make best use of R.

The S language, of which R is a dialect, has three object systems, known informally as S3, S4 anf RC. Their names originate from the version of S they appear first. S3 objects, classes and methods have been available in R from the beginning. S4 objects, classes and methods have been available in R through the `methods` package, attached by default since R version 1.7.0.

7.2 S3

S3 objects, classes and methods have been available in R from the beginning, they are informal, yet ‘’very interactive’‘. S3 was first described in the’‘White Book” (Statistical Models in S).

S3 is not a real class system, it mostly is a set of naming conventions. Classes are attached to objects as simple attributes. Method dispatch looks for the class of the first argument and then searches for functions conforming to a naming convention: `do()` methods for objects of class `obj` are called `do.obj()`. If no `do` method is found, S3 searches for `do.default()`.

This system is simple and powerful at the same time. Objects of widely used classes such as `lm` or `glm` are still implemented as S3:

```
showMethods("lm")
```

```
##
## Function "lm":
##  <not an S4 generic function>
```

Nevertheless, S3 is far from be structured and validated:

```
f <- function(x) {
  x <- list(x)
  class(x) <- "lm"
  x
}

f(x ="Mickey Mouse")
```

```
##
## Call:
## NULL
##
## No coefficients
```

The system should not accept that a simple string can be defined as an object of class linear model.

7.3 S4

S4 objects, classes and methods are much more formal and rigorous, hence ''less interactive''. S4 was first described in the''Green Book'' (Programming with Data). In R it is available through the methods package, attached by default since version 1.7.0.

7.3.1 Example: Class rectangle

As a simple example, we can consider a class rectangle. As any rectangle can be entirely defined by the dimensions of its sides, a class for objects of type rectangle can be defined as made of two numeric slots: x and y representing the sides of the rectangle.

```
setClass("rectangle",
  representation (x= "numeric", y = "numeric"),
  prototype(x = 1, y = 1)
)
```

Note the use of argument prototype with function setClass. This argument allows to create a rectangle of sides x=1 and y=1 whenever its dimensions are not explicitelly given.

Once the class is defined, an object of class rectangle can be created by:

```
new("rectangle")

## An object of class "rectangle"
## Slot "x":
## [1] 1
##
## Slot "y":
## [1] 1

new("rectangle", x = 2, y = 4)

## An object of class "rectangle"
## Slot "x":
## [1] 2
##
## Slot "y":
## [1] 4

new("rectangle", x = -3, y = 5)

## An object of class "rectangle"
## Slot "x":
## [1] -3
##
## Slot "y":
## [1] 5
```

Generally, objects are not created directly by using function `new()`. We usually define a specific function in order to perform this task:

```
rectangle <- function (x, y) {
  if (!"x" %in% names(match.call()) & !"y" %in% names(match.call())) {
  rectangle <- new("rectangle")}
  else if (!"x" %in% names(match.call())) {rectangle <- new("rectangle", y = y)}
  else if (!"y" %in% names(match.call())) {rectangle <- new("rectangle", x = x)}
  else rectangle <- new("rectangle", x = x, y = y)
  rectangle
}
```

The prototype argument of class definition allows great flexibility when passing arguments to function rectangle:

```
rectangle(x = 2, y = 7)

## An object of class "rectangle"
## Slot "x":
## [1] 2
##
## Slot "y":
## [1] 7

rectangle(x = 2)

## An object of class "rectangle"
## Slot "x":
## [1] 2
##
## Slot "y":
## [1] 1

rectangle(y = 2)

## An object of class "rectangle"
## Slot "x":
## [1] 1
##
## Slot "y":
## [1] 2

rectangle()

## An object of class "rectangle"
## Slot "x":
## [1] 1
##
## Slot "y":
## [1] 1
```

```
rectangle(x = -2, y = 0)

## An object of class "rectangle"
## Slot "x":
## [1] -2
##
## Slot "y":
## [1] 0

new("rectangle", x = "three", y = 2)

## Error: invalid class "rectangle" object: invalid object for slot "x" in
## class "rectangle": got class "character", should be or extend class
## "numeric"
```

As seen, class definition performs same validity check by itself. Nevertheless, either zeros or negative numbers should not be accepted as valid input for sides dimensions. For appropriate validity check a specific validity method can be defined by using function `setValidity()`. Note that validity methods are stored together with class definitions.

```
setValidity("rectangle",
  function(object) {object@x > 0 & object@y > 0}
)

## Class "rectangle" [in ".GlobalEnv"]
##
## Slots:
##
## Name:        x       y
## Class: numeric numeric
```

Testing the class after validity method is defined allows great control on input arguments:

```
new("rectangle", x = -3 , y = 2)

## Error: invalid class "rectangle" object: FALSE

rectangle(x = -2, y = 1)

## Error: invalid class "rectangle" object: FALSE

new("rectangle", x = "three", y = 2)

## Error: invalid class "rectangle" object: invalid object for slot "x" in
## class "rectangle": got class "character", should be or extend class
## "numeric"
```

```
rectangle(x = "three", y = 2)

## Error: invalid class "rectangle" object: invalid object for slot "x" in
## class "rectangle": got class "character", should be or extend class
## "numeric"
```

After the class is defined, we can define basic methods, generally: `show`, `print`, `summary` and `plot`. Method `show()` is usually the first method we develop as this method is applied when objects are called without a function and allows objects to be displayed in a ordered and clear fashion.

```
setMethod(f = "show", signature = "rectangle",
  definition <-  function(object) {
    x <- object@x ; y <- object@y
    cat(class(object), "of side x =", x , "and side y =",
      y , "\n")
   invisible(NULL)
})

## [1] "show"

r42 <- rectangle(4,2)
show(r42)

## rectangle of side x = 4 and side y = 2

r42

## rectangle of side x = 4 and side y = 2
```

We can define a method `print()`, with identical output to `show()`:

```
setMethod(f = "print", signature = "rectangle",
  definition = function(x) {
    object <- x
    x <- object@x ; y <- object@y
    cat(class(object), "of side x =", x , "and side y =",
      y , "\n")
    invisible(NULL)
})

## Creating a generic function for 'print' from package 'base' in the global environment

## [1] "print"

r27 <- rectangle(2,7)
print(r27)
```

```
## rectangle of side x = 2 and side y = 7
```

We can write a more exhaustive output with method `summary()`:

```
setMethod(f = "summary", signature = "rectangle",
  definition = function(object) {
    x <- object@x ; y <- object@y
    perimeter <- 2*x+2*y
    area <- x*y
    print(object)
    cat("Perimeter =" , perimeter , "\n")
    cat("Area =" , area, "\n")
    invisible(list (sides = c(x, y),
      perimeter = perimeter, area = area))
})
```

```
## Creating a generic function for 'summary' from package 'base' in the global environment
```

```
## [1] "summary"
```

```
r42 <- rectangle(4, 2)
summary(r42)
```

```
## rectangle of side x = 4 and side y = 2
## Perimeter = 12
## Area = 8
```

```
r42_area_perimeter <- summary(r42)
```

```
## rectangle of side x = 4 and side y = 2
## Perimeter = 12
## Area = 8
```

```
r42_area_perimeter
```

```
## $sides
## [1] 4 2
##
## $perimeter
## [1] 12
##
## $area
## [1] 8
```

`area` and `perimeter`, as they have been computed are returned as invisible from method `summary()`.

Method `plot()` closes the list of standard methods usually developed for any class:

```
setMethod(f = "plot", signature = "rectangle",
  definition =  function(x, y, col = "lightgray" ,
  border = "black", xlab = "x", ylab = "y", ...) {
    object <- x
    x <- object@x ; y <- object@y
    d <- max(c(x, y))
    plot(c(0, d, d, 0), c(0, 0, d , d ),
      type = "n", asp = 1,
      xlab = xlab , ylab = ylab, ...)
    polygon (c(0, x, x, 0), c(0, 0, y, y),
     col = col, border = border)
    grid()
    invisible(NULL)
})
```

```
## Creating a generic function for 'plot' from package 'graphics' in the global environment

## [1] "plot"
```

```
r42 <- rectangle(4, 2)
plot(r42)
```

`print()`, `plot()` and `summary()` are existing generic methods. If required, we can define a new generic method. For instance, a `rotate()` method that rotates the rectangle of `90` degree can be defined in two steps:

- Define the generic `rotate()` method as it does not exists by default in `R`.
- Define a specific `rotate()` method for objects of class rectangle.

```
setGeneric("rotate",
  function(object, ...)  standardGeneric("rotate")
)
```

```
## [1] "rotate"
```

Given the rotate generic method a rotate specific method for class rectangle can be written as:

```
setMethod(f = "rotate", signature = "rectangle",
  definition = function(object) {
    xx <- object@x
    object@x <- object@y
    object@y <- xx
    object
})
```

```
## [1] "rotate"
```

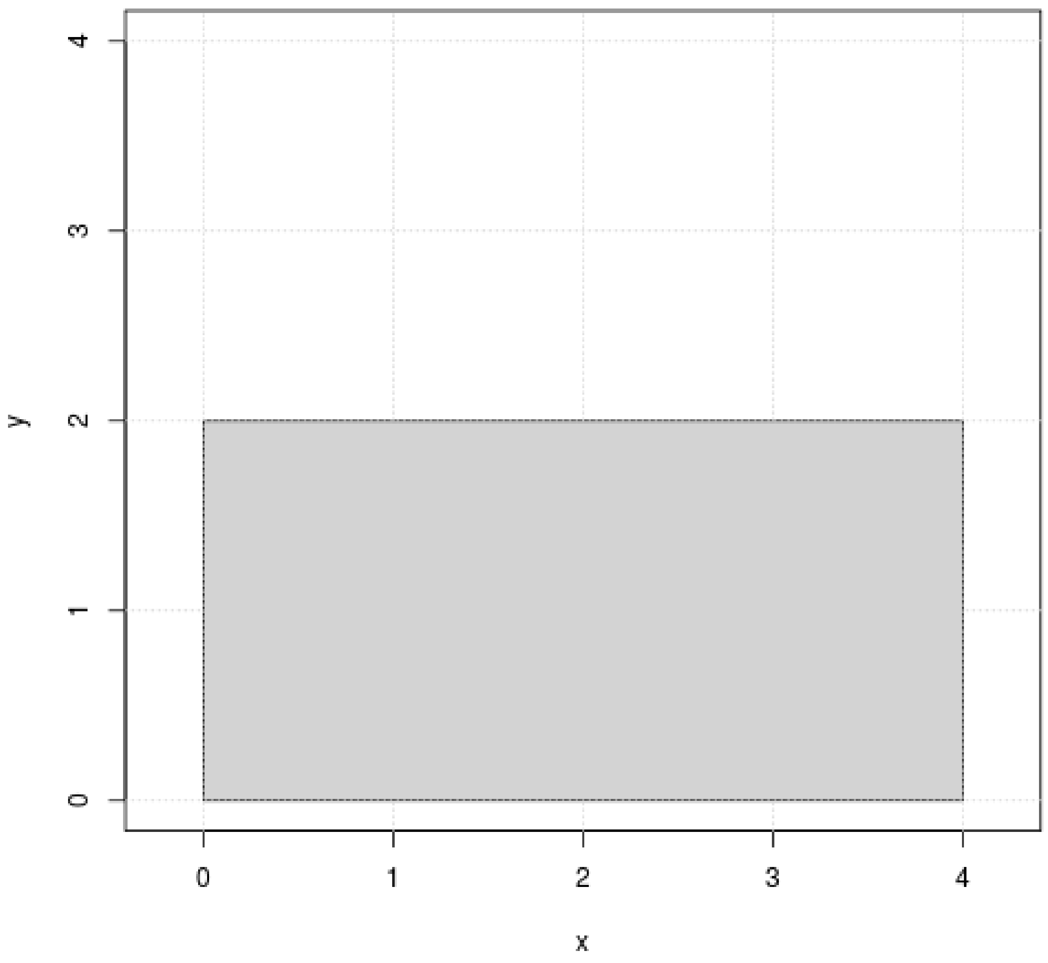


Figure 7.1: plot of chunk s4-018

```
r12 <- rectangle(1,2)
r21 <- rotate(r12)
par(mfrow = c(1,2))
plot(r12, col = "darkred")
plot(r21 , col = "darkblue")
```

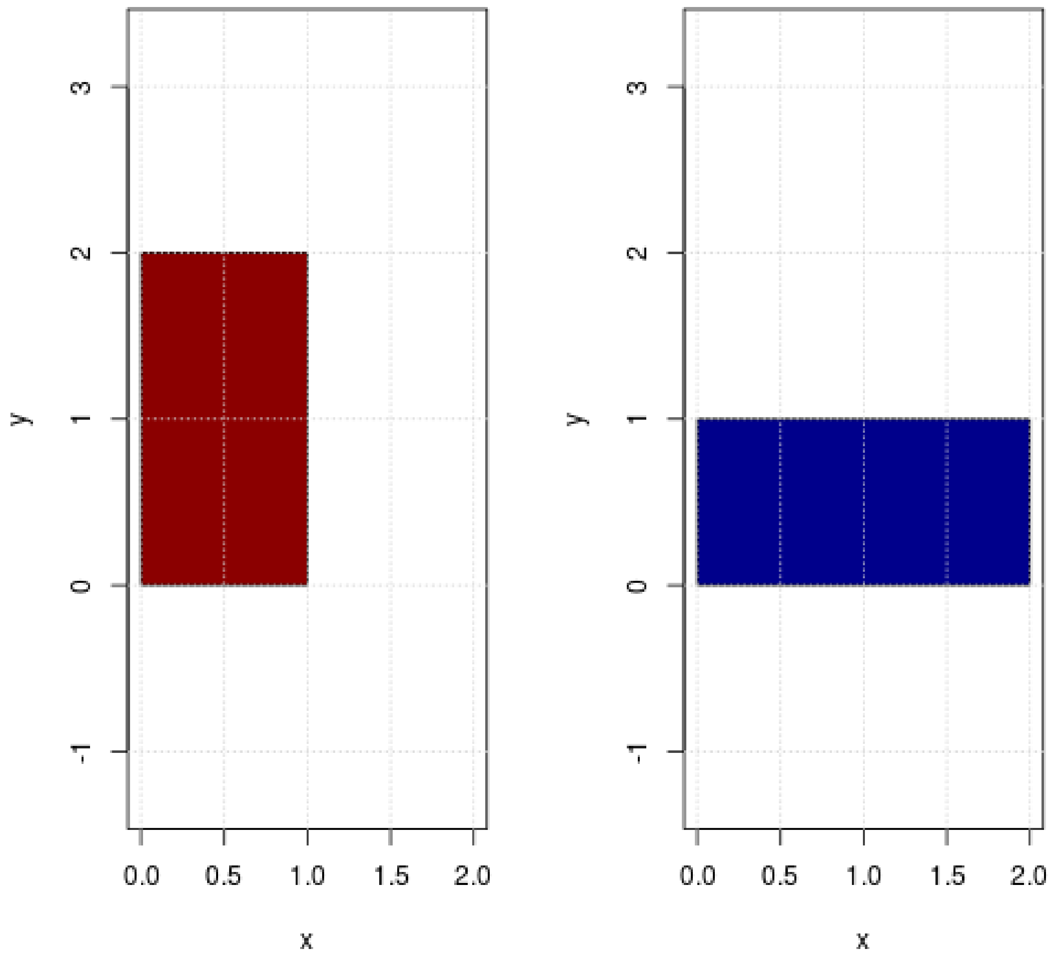


Figure 7.2: plot of chunk s4-020

```
par(mfrow = c(1,1))
```

7.3.2 Example: Class parallelepiped

Given class `rectangle`, class `parallelepiped` can be defined as an extension of class rectangle. The whole structure of class `rectangle` is inherited by `parallelepiped`. Therefore, when defining the new class, only additional slots need to be defined. Specifically, only slot `z` representing the third dimension of the parallelepiped needs to be defined. Slots `x` and `y` are implicitly inherited from parent class `rectangle` along with all defined methods.

```
setClass("parallelepiped",
  representation (z = "numeric"),
  prototype(z = 1),
  contains = "rectangle"
)

new("parallelepiped")

## parallelepiped of side x = 1 and side y = 1
```

Class `parallelepiped` is explicitly defined as an extension of class rectangle and `R` tracks all of this within the definitions of both `rectangle` and `parallelepiped` classes.

```
getClass("parallelepiped")

## Class "parallelepiped" [in ".GlobalEnv"]
##
## Slots:
##
## Name:        z       x       y
## Class: numeric numeric numeric
##
## Extends: "rectangle"

getClass("rectangle")

## Class "rectangle" [in ".GlobalEnv"]
##
## Slots:
##
## Name:        x       y
## Class: numeric numeric
##
## Known Subclasses: "parallelepiped"
```

Specific methods can be written for class `parallelepiped`. Alternatively, methods of the parent class `rectangle` are used. Note that this may lead to some confusion:

```
prl <- new("parallelepiped")
print(prl)
```

```
## parallelepiped of side x = 1 and side y = 1
```

Clearly these are not all the information someone would expect about a parallelepiped. A new print method should be written that includes, at least, side `z`:

```
setMethod(f = "print", signature = "parallelepiped",
  definition = function(x) {
    object <- x
    x <- object@x ; y <- object@y ; z <- object@z
    cat(class(object), "of sides x =", x ," y =",y , " z =" , z, "\n")
    invisible(NULL)
})
```

```
## [1] "print"
```

```
print(prl)
```

```
## parallelepiped of sides x = 1  y = 1  z = 1
```

7.3.3 Example: Class square

The same mechanism can be used the other way round in order to define classes that are specific cases of an existing class. Again, methods are inherit from parent to child:

```
setClass("square",
  contains = "rectangle"
)

square <- function(x) {
  y <- x
  new("square", x = x, y = y)
}

s4 <- square(4)
print(s4)
```

```
## square of side x = 4 and side y = 4
```

```
summary(s4)
```

```
## square of side x = 4 and side y = 4
## Perimeter = 16
## Area = 16
```

```
plot(s4)
```

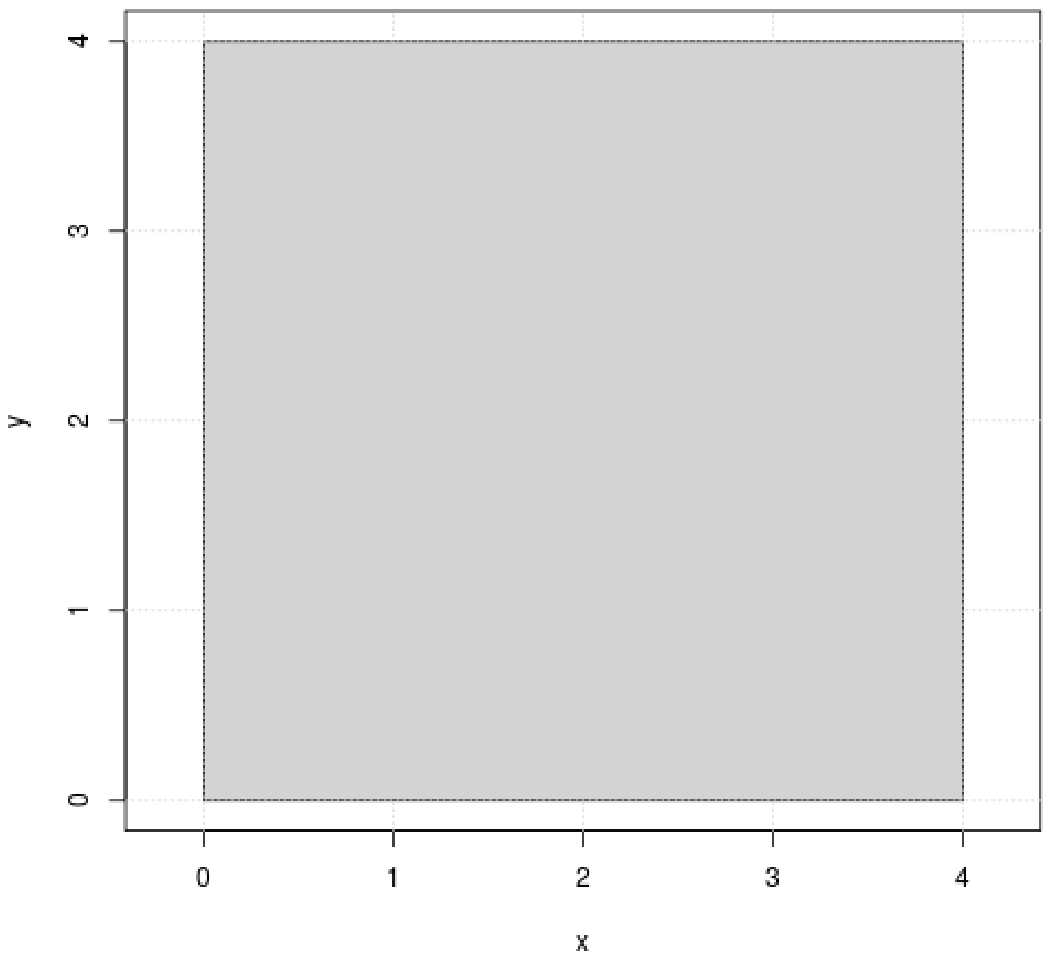


Figure 7.3: plot of chunk s4-025

Moreover, a class `square` can be defined as a coerced class from class `rectangle` by writing a `definition` for function `setAs()`. As an example, `definition` may impose that any `rectangle(x, y)` is coerced into a `square(x)`.

Definition written within `setAs()` function is then used by `R` when calling function `as()`:

```
setAs(from = "rectangle", to = "square",
  def = function(from) {
    square = square(x = from@x)
    square
})

r35 <- rectangle(3, 5)
s33 <- as(r35, "square")
s33

## square of side x = 3 and side y = 3
```

7.3.4 Example: "Rolygons" S4 with closures

The combination of the S4 methods with functional programmimg tecniques permits the development of quite interesting coding techniques.

In this case we want to generate a set of functions each of them returning a regular polygon: square, pentagon, etc with a built in `plot` method.

Thus, we first define a `rolygon()` function that returns a generic function capable of generating specific regular polygons with plot method inherited from rolygons environment:

```
rolygon <- function(n){

    # Define rolygon class
    setClass("rolygon", representation(n = "numeric", s = "numeric"))

    # Define a plot method for object of class rolygon
    setMethod(f = "plot", signature = "rolygon",
              definition = function(x, y){
                object <-  x
                s <-  object@s ; n = object@n
                pi <- base::pi
                rho <-  (2*pi)/n
                h <-  .5*s*tan((pi/2)-(pi/n))
                r <-  sqrt(h^2+(s/2)^2)
                sRho <-  ifelse( n %% 2 == 0 , (pi/2- rho/2)  , pi/2)
                cumRho <-  cumsum(c(sRho, rep(rho, n)))
                cumRho <-  ifelse(cumRho > 2*pi, cumRho-2*pi, cumRho)
                x <-  r*cos(cumRho)
                y <-  r*sin(cumRho)
                par(pty = "s")
                plot(x, y, type = "n", xlab = "", ylab = "")
                lines(x, y, col = "red", lwd = 2)
                points(0,0, pch = 16, col = "red")
                grid()
                invisible(NULL)
              })

    # Define a function that returns an object of class rolygon
    f <- function(s){new("rolygon", n = n, s = s)}

    # Return the newly created function
    return(f)
  }
```

Note that class `rolygon`, its `plot` method and `f()` function are all defined within the evaluation environment of `rolygon()`. When `rolygon` is evaluated, `f()` is returned and `f()` remembers about class `rolygon` and its plotting method.

As a result, we can define an `heptagon()` function as:

```
heptagon <- rolygon(n = 7)
```

a specific heptagon of side = 1 becomes:

```
e1 <- heptagon(1)
```

as `heptagon()` has a plot method built in, we only need:

```
plot(e1)
```

Finally, with a bit of imagination:

```
circumference <- rolygon(n = 10^4)

plot(circumference(s = base::pi/10^4))
```

7.3.5 S4 House keeping

Package `methods` dispatches several function for S4 object oriented programming and most of them have already been illustrated in the previous section:

- define classes: `setClass()`
- create objects: `new()`
- define generics: `setGeneric()`
- define methods: `setMethods()`

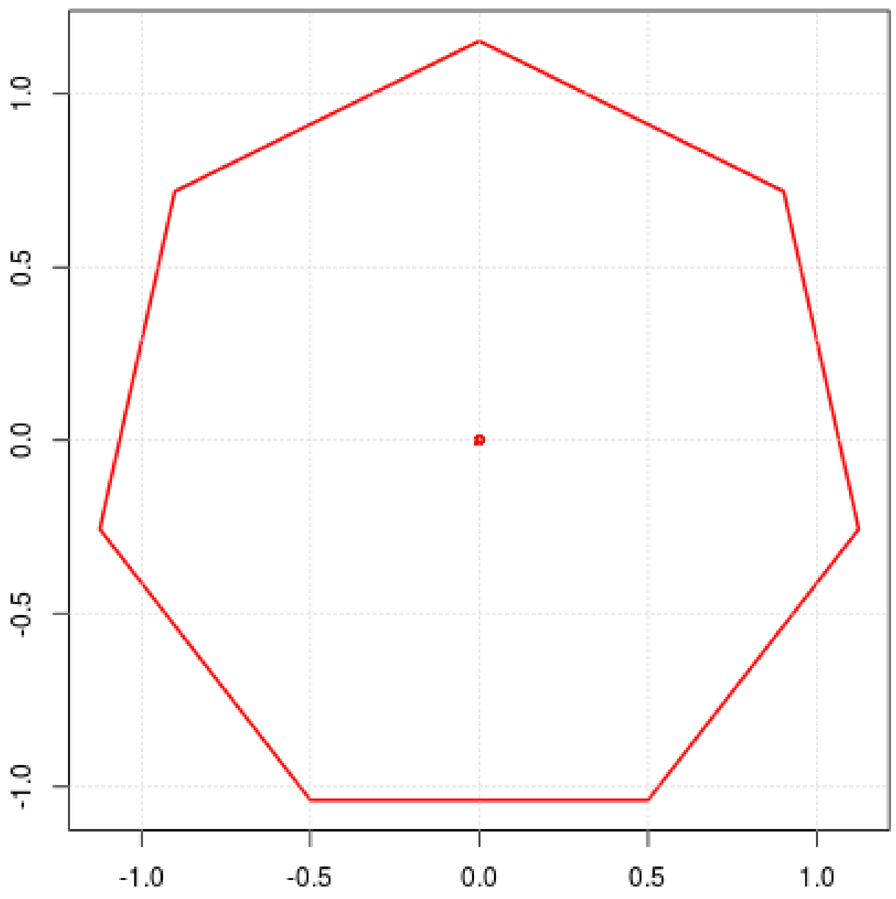


Figure 7.4: plot of chunk s4-030

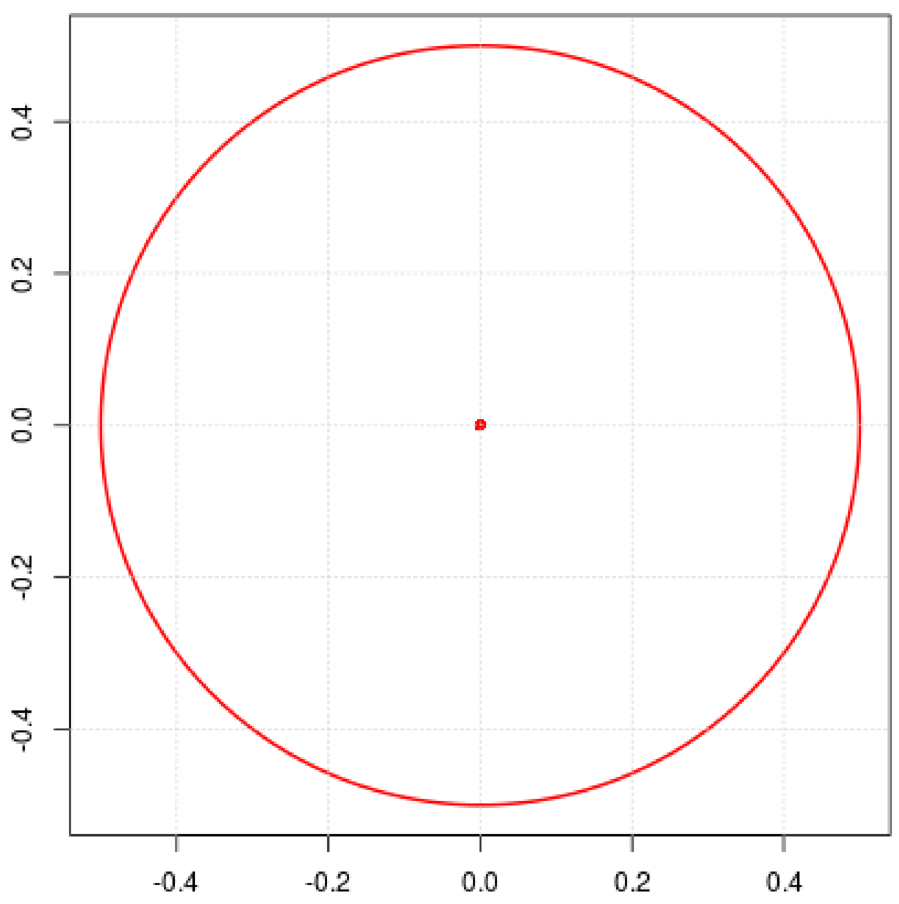


Figure 7.5: plot of chunk s4-032

- delete classes: `removeClasses()`
- delete methods: `removeMethods()`
- convert objects: `as()`, `setAs()`
- check object validity: `setValidity()`, `validObject()`
- access registry: `showClass()`, `showMethods()`, `getMethod()`

When a class or a method is created, `R` saves it in a dedicated registry within the working environment. Each package has its own dedicated registry. Methods and classes are usually accessed by dedicated functions. Functions `showClasses()` and `getClasses()` return the structure, of a class. For instance, in order to gain the structure of class rectangle:

```
showClass("rectangle")
```

```
## Class "rectangle" [in ".GlobalEnv"]
##
## Slots:
##
## Name:        x       y
## Class: numeric numeric
##
## Known Subclasses: "parallelepiped", "square"
```

```
getClass("rectangle")
```

```
## Class "rectangle" [in ".GlobalEnv"]
##
## Slots:
##
## Name:        x       y
## Class: numeric numeric
##
## Known Subclasses: "parallelepiped", "square"
```

The validity function, if defined, of a given class is obtained by function `getValidity()`:

```
getValidity(getClass("rectangle"))
```

```
## function(object) {object@x > 0 & object@y > 0}
```

The function `showMethods()` checks weather a method exists for given class; to check `show` and `print` methods for class `rectangle`:

```
showMethods(f = c("show", "print"), classes = "rectangle")
```

```
## Function: show (package methods)
## object="rectangle"
##
## Function: print (package base)
## x="rectangle"
```

Note that omitting argument `f` within `showMethods()` returns all methods for a given class:

```
showMethods(classes = "rectangle")
```

```
## Function: coerce (package methods)
## from="rectangle", to="square"
## from="square", to="rectangle"
##
## Function: initialize (package methods)
## .Object="rectangle"
##     (inherited from: .Object="ANY")
##
## Function: plot (package graphics)
## x="rectangle"
##
## Function: print (package base)
## x="rectangle"
##
## Function: rotate (package .GlobalEnv)
## object="rectangle"
##
## Function: show (package methods)
## object="rectangle"
##
## Function: summary (package base)
## object="rectangle"
```

The definition of a given method can be displayed by:

```
getMethod("print", "rectangle")
```

```
## Method Definition:
##
## function (x, ...)
## {
##     .local <- function (x)
##     {
##         object <- x
##         x <- object@x
##         y <- object@y
##         cat(class(object), "of side x =", x, "and side y =",
##             y, "\n")
##         invisible(NULL)
##     }
##     .local(x, ...)
## }
##
## Signatures:
```

```
##         x
## target  "rectangle"
## defined "rectangle"
```

As methods and classes are created they can be deleted respectively with functions `removeClasses()` and `removeMethods()`.

7.4 References

kindly provided at : stackoverflow ### On the web

- The methods help files : help files from the package methods, where much of the necessary information can be found
- S4 classes in 15 pages : Short introduction on the programming with S4 objects.
- How S4 methods work : more explanation about the underlying mechanisms.
- Not so short introduction to S4 : with practical examples of how to construct the classes and some useful tips. It contains a handy overview as appendix, but contains errors as well. Care should be taken using this one.
- OOP in R : handout notes with practical examples on S3 and S4
- S4 Objects : presentation by Thomas Lumley about S4 objects

7.4.1 Books

- Software for Data Analysis-Programming with R (J. Chambers) : A classic, although not reviewed positive everywhere, that contains a large section on S4
- R programming for Bioinformatics (R. Gentleman) : specifically directed towards working with Bioconductor, which is completely based on S4. But it gives a broad overview and is useful for many other people too.

7.5 RC reference classes

A recent development in R is *Reference classes* also known as `RC` or `R5`.

`RC` makes `R` object oriented programming paradigm very close to those implemented in `C++` or `Java`.

On the other hand, when approaching reference classes we should also take into account that:

- Documentation on reference classes is still very limited
- RC require to learn a new form of programming syntax
- Mutable state does not fit very well the *no side effect* nature of most `R` functions

7.5.1 Example: `zero_one`, a toy example

As first basic example consider creating a new class `zero_one` with two self explicative methods associated to it: `$set_to_zero()` and `set_to_one()`

```
zero_one  <- setRefClass("zero_one",
                      fields = list( x = "numeric"),
                      methods = list(
                        set_to_zero = function(x){
                          x <<- 0
                        },
                        set_to_one = function(x){
                          x <<- 1
                        }
                      )
)
```

First notice that:

- `RC` does not simply register a class, as `setClass()` in `S4` does, but holds the newly created class in an object. Now, object `zero_one` holds class `zero_one`.
- `fields` corresponds to representation in `S4`
- `methods` defines functions as true methods belonging to the class

The call to `setRefClass()` defines class `zero_one` and returns a `generator` object for class `zero_one`.

By using method `$new()` we create a new object of class `zero_one`

```
zero_one_test <- zero_one$new(x = 33)
zero_one_test
```

```
## Reference class object of class "zero_one"
## Field "x":
## [1] 33
```

We can now apply methods `$set_to_zero()` and `set_to_one()` to the newly created object:

```
zero_one_test$set_to_zero()
```

and see how `zero_one_test` modifies its fields

```
zero_one_test
```

```
## Reference class object of class "zero_one"
## Field "x":
## [1] 0
```

R functions usually do not have any side effects. Objects are modified by assignment and this happens within a *copy on mofy* criterion. Reference classes instead allows us to mutate the state of objects without duplicating them.

Reference class methods can use the operator `<<-`. This modifies the value of a field in place by using a combination of `environment` and `makeActiveBinding()`.

7.5.2 Example: A stack implementation

Within this example we define a `stack` implementation with methods `$put_in()` and `$get_out()` where the latest come with two flavors:

- `fifo`: first in first out
- `lilo`: last in last out

We first define the reference class:

```
stack  <- setRefClass("stack",
                      fields = list( stack = "numeric"),
                      methods = list(
                        put_in = function(x){
                          stack <<- c(stack, x)
                        },
                        get_out = function(n = 1 , method = "fifo"){
                          stopifnot(method %in% c("fifo", "lilo"))
                          if(method == "fifo"){
                            first <- 1:n
                            stack <<- stack[-first]
                          }
                          if(method == "lilo"){
                            N <- length(stack)
                            last <- c((N-n+1):N)
                            stack <<- stack[-last]

                          }
                        }
                      )
)
```

And now we test it:

```
stack_test <-stack$new(stack = 0)
stack_test$put_in(1:10)
stack_test$get_out(method = "fifo", n = 2)
stack_test

## Reference class object of class "stack"
## Field "stack":
## [1]  2  3  4  5  6  7  8  9 10
```

```
stack_test$get_out(method = "lilo", n = 2)
stack_test

## Reference class object of class "stack"
## Field "stack":
## [1] 2 3 4 5 6 7 8
```

Chapter 8

Debugging and Profiling

8.1 Profiling

Profiling `R` code helps to identify bottlenecks and pieces of code that needs are not efficiently implemented.

Profiling our code is usually the last thing we do when developing functions and packages. With serious profiling, the amount of time required can be drastically reduced with very simple changes to our code.

Benchmarking `R` code is about comparing performances of different solutions to the same problem in terms of execution time.

Both techniques aim to reduce computational time.

8.1.1 `Rprof`

As a main tool for profiling `R` code we make use of function `Rprof()`.

As a basic example to demonstrate `Rprof()` capabilities we consider a trivial function `f1()` written a inefficient `for()` loop fashion:

```
f1 <- function(x, s1 = 1 , s2 = 2){
  n <- length(x)
  y <- NULL
  for ( i in 1:n){
    if (x[i] %% 2 == 0 ) tmp <- x[i]+rnorm(1, 0 , s2)
    else tmp <- x[i]+rnorm(1, 0, s1)
    y <- c(y, tmp)
  }
  sum(y)
}
f1(x = 1:5)
```

```
## [1] 17.33
```

When running `f1()` on a large vector `x` it may require a significant amount of computing time:

```
system.time(f1(x = 1:10^5))
```

```
##    user  system elapsed
##   26.27    1.34   27.61
```

In order to understand if any bottle neck exists within `f1()` we may want to **profile** function `f1()` by using function `Rprof()`.

Function `Rprof()` is used to control profiling. Profiling works by recording at fixed intervals, by default every 20 msecs, which line in which `R` function is being used and records the results in a file passed as argument to `Rprof()`. Function `summaryRprof()` can then be used to summarize the activity.

```
Rprof("f1.Rprof")
f1(x = 1:10^5)
```

```
## [1] 5e+09
```

```
Rprof(NULL)
summaryRprof("f1.Rprof")$by.self
```

```
##               self.time self.pct total.time total.pct
## "c"               25.52    96.67      25.52     96.67
## "f1"               0.34     1.29      26.40    100.00
## ".External"        0.34     1.29       0.34      1.29
## "rnorm"            0.08     0.30       0.42      1.59
## "%%"               0.08     0.30       0.08      0.30
## "=="               0.02     0.08       0.02      0.08
## "+"                0.02     0.08       0.02      0.08
```

We can observe that function `c()` takes 25.52 seconds corresponding to 96.67 per cent of the total execution time. This is because of the *wrong* usage we are making of function `c()` within `f1()` when computing `y <- c(y, tmp)`. At each iteration `R` is forced to copy vector `y` in a larger memory space to allocate the longer vector.

We can avoid this multiple copying of `y` by simply defining in advanced the length of vector `y`:

```
f2 <- function(x, s1 = 1 , s2 = 2){
  n <- length(x)
  y <- numeric(n)
  for ( i in 1:n){
    if (x[i] %% 2 == 0 ) tmp <- x[i]+rnorm(1, 0 , s2)
    else tmp <- x[i]+rnorm(1, 0, s1)
    y[i] <- tmp
  }
  sum(y)
}
f2(x = 1:5)
```

```
## [1] 7.112
```

and, if we run `Rprof()` again:

```
Rprof("f2.Rprof")
f2(x = 1:10^5)
```

```
## [1] 5e+09
```

```
Rprof(NULL)
summaryRprof("f2.Rprof")$by.self
```

```
##              self.time self.pct total.time total.pct
## ".External"       0.38    51.35       0.38     51.35
## "f2"              0.22    29.73       0.74    100.00
## "rnorm"           0.12    16.22       0.50     67.57
## "%%"              0.02     2.70       0.02      2.70
```

we can observe that the execution time is now down to 0.74 seconds and that `rnorm()` is taking a sensible amount or time.

In case we want to evaluate the gain in efficiency by using `lapply()` instead of a `for()` loop we could rewrite `f2()` as:

```
f3 <- function(x, s1 = 1 , s2 = 2){
  f31 <- function(x, s1 , s2){
    ifelse(x %% 2 == 0, x+rnorm(1, 0 , s2) , x+rnorm(1, 0, s1))
  }
sum(vapply(x, f31, FUN.VALUE=numeric(1), s1, s2 ))
}
f3(x = 1:5)
```

```
## [1] 17.88
```

and profile this functions as usual by:

```
Rprof("f3.Rprof")
f3(x)
```

```
## Error: object 'x' not found
```

```
Rprof(NULL)
summaryRprof("f3.Rprof")$by.self
```

```
## [1] self.time  self.pct   total.time total.pct
## <0 rows> (or 0-length row.names)
```

Note that in this case, function `f2()` takes 0.74 seconds while function `f3()` requires 0 seconds.

We can make full use of `R` vectorized capabilities and write:

```
f4 <- function(x, s1 = 1 , s2 = 2){
  n <- length(x)
  s1 <- rnorm(n, 0 , s1)
  s2 <- rnorm(n, 0 , s2)
  sum(ifelse(x %% 2 == 0, x+s1 , x+s2))
}
f4(x = 1:5)
```

```
## [1] 20.81
```

We can now profile this functions as usual by:

```
Rprof("f4.Rprof")
f4(x)
```

```
## Error: object 'x' not found
```

```
Rprof(NULL)
summaryRprof("f4.Rprof")$by.self
```

```
## [1] self.time  self.pct   total.time total.pct
## <0 rows> (or 0-length row.names)
```

We see can see no specific bottle necks and a very fast function:

```
sapply(paste("f", 1:4, ".Rprof", sep = ""), function(x) summaryRprof(x)$sampling.time)
```

```
## f1.Rprof f2.Rprof f3.Rprof f4.Rprof
##    26.40     0.74     0.00     0.00
```

Finally, as a whole, function `summaryRprof()` returns a list of four components:

```
names(summaryRprof("f2.Rprof"))
```

```
## [1] "by.self"         "by.total"        "sample.interval" "sampling.time"
```

Table `$by.self` lists the time spent by functions alone, while the table `$by.total` lists the time spent by functions and all the functions they call. In the `f2()` simple case, function `f2()` itself just takes 0.22 seconds to do things like:

- initialize its own execution environment
- filling it up with promises arguments
- starting the execution of its body

while the same function remains active until the end of the execution: 0.74 seconds. The lines marked with `.External` refer to calls to external `C` or `Fortran` code.

8.1.1.1 Rprof with memory.profiling

Profiling can be used to gain information about timing as well as memory. In order to gain memory usage we need to call `Rprof()` with `memory.profiling` enabled and calling `summaryRprof()` with `memory = "both"`:

```
Rprof("f1.Rprof", memory.profiling = TRUE)
f1(x = 1:10^5)

## [1] 5e+09

Rprof(NULL)
summaryRprof("f1.Rprof", memory = "both")$by.self

##              self.time self.pct total.time total.pct mem.total
## "c"              26.22    96.33      26.22     96.33     807.3
## ".External"       0.44     1.62       0.44      1.62      17.7
## "f1"              0.38     1.40      27.22    100.00     839.2
## "%%"              0.08     0.29       0.08      0.29       1.4
## "rnorm"           0.06     0.22       0.50      1.84      20.5
## "=="              0.02     0.07       0.02      0.07       0.0
## "+"               0.02     0.07       0.02      0.07       0.0
```

In this case a column indicating the memory, expressed in `Mb`, as used by each function, is added to the previous output.

Note that `memory.profiling` requires `R` to be compiled with `--enable-memory-profiling`. This option should be enabled by default under Windows and Mac-OS but not on all Linux distributions.

8.2 Benchmarking

Function `Rprof()` represent a great tool when investigating for bottle necks within functions in order to remove them and speed up our code.

At the same time we may want to compare different solutions to understand in order to understand if any of them may provide any real advantage over the others.

8.2.1 rbenchmark

```
require(rbenchmark)
```

```
## Loading required package: rbenchmark
```

Library `rbenchmark` is intended to facilitate benchmarking of arbitrary R code.

The library consists of just one function, `benchmark()`, which is a simple wrapper around `system.time()`.

Given a specification of the benchmarking process: counts of replications, evaluation environment and an arbitrary number of expressions, benchmark evaluates each of the expressions in the specified environment, replicating the evaluation as many times as specified, and returning the results conveniently wrapped into a data frame.

As a first case we can compare the differece in performaces between `sapply()` and `vapply()` when applied to a simple mathematical function:

```
s <- seq(-pi, pi, len = 1000)
f <- function(x) sin(x)/ (1-cos(x))
benchmark(vapply(s,f, FUN.VALUE = numeric(1)), sapply(s,f))
```

```
##                                  test replications elapsed relative
## 2                        sapply(s, f)          100   0.184    1.122
## 1 vapply(s, f, FUN.VALUE = numeric(1))          100   0.164    1.000
##   user.self sys.self user.child sys.child
## 2     0.184        0          0         0
## 1     0.163        0          0         0
```

and observe the gain in performances of `vapply()` compared with `sapply()`

As a second example, we can consider the calculation of the 95% quantile of the distribution of the correlation coefficient between two `N(0,1)` vectors of given sizes `n=10`.

As a first case we define a function `f_loop()` by using a simple for loop over `k` iterations.

```
f_loop <- function(n = 10, k = 1e+05){
  z = numeric(k)
  for (i in 1:k) {
    x = rnorm(n, 0, 1)
    y = rnorm(n, 0, 1)
    z[i] = cor(x, y)
  }
quantile(z, 0.95)
}
```

As a second case we develop the same function using a functional approach based on `replicate()`

```
f_rep <- function(n = 10, k = 1e+05){
  z <- replicate(k, cor(rnorm(n), rnorm(n)))
  quantile(z, .95)
}
```

As a third case we introduce a function `matrix()` within `replicate()`:

```
f_mat <- function(n = 10, k = 1e+05){
  z <- replicate(k, cor(matrix(rnorm(n*2), nrow = n, ncol = 2))[1,2])
  quantile(z, .95)
}
```

Once the three functions have been defined we can compare them by using `rbenchmark()`

```
benchmark(f_loop(), f_rep(), f_mat(), replications = 10, order = "elapsed")
```

```
##       test replications elapsed relative user.self sys.self user.child
## 3  f_mat()           10   58.81    1.000     58.80    0.004          0
## 1 f_loop()           10   61.63    1.048     61.63    0.005          0
## 2  f_rep()           10   64.65    1.099     64.64    0.004          0
##   sys.child
## 3         0
## 1         0
## 2         0
```

We can see that all functions, despite the different programming styles, have very similar performances.

8.2.2 `microbenchmark`

In some cases we may want to benchmark two or more solutions that require a very small amount of time to perform but are very often used in our code.

As as an example, we may want to compare `[i]` and `[[i]]` when applied to atomic objects with `i` being a single integer.

First, let's notice that `x[i]` and `x[[i]]` return the same result when `i` is a single integer:

```
x <- 1:100
i <- 66
x[i]
```

```
## [1] 66
```

```
x[[i]]
```

```
## [1] 66
```

In principle, in order to evaluate performances of these functions, we could use:

```
benchmark(x[i], x[[i]], replications = 1000)
```

```
##     test replications elapsed relative user.self sys.self user.child
## 2 x[[i]]         1000   0.002        1     0.003        0          0
## 1   x[i]         1000   0.002        1     0.003        0          0
##   sys.child
## 2         0
## 1         0
```

but a better choice is represented by library `microbenchmark`:

```
library(microbenchmark)
test <- microbenchmark(x[i] , x[[i]], times = 1000)
test
```

```
## Unit: nanoseconds
##    expr min  lq  mean median    uq   max neval
##    x[i] 127 137 221.4    150 213.0 13217  1000
##  x[[i]] 168 180 265.3    190 255.5 20052  1000
```

Note that timing is reported in `nonoseconds` and that the output is made of several statistics.

Moreover, we can use the graphics facilities provided by library `ggplot2` to compare the two distributions:

```
library(ggplot2)
```

```
## Loading required package: methods
```

```
autoplot(test)
```

8.3 Debugging

Debugging is typically what programmers do about 90% of the time.

This is a sad but not unrealistic fact of life. Given that fact, the creators of R have generously provided useful debugging tools to make programmers' lives a little easier.

The debugging tools should be used as much as necessary to minimize the time spent debugging and to maximize the time spent, as John Chambers wrote, *"turning ideas into software"*.

However, it is all to easy for a programmer to develop an unhealthy relationship with his debugger. This is to be avoided. The debugger should not replace common sense in programming and careful design.

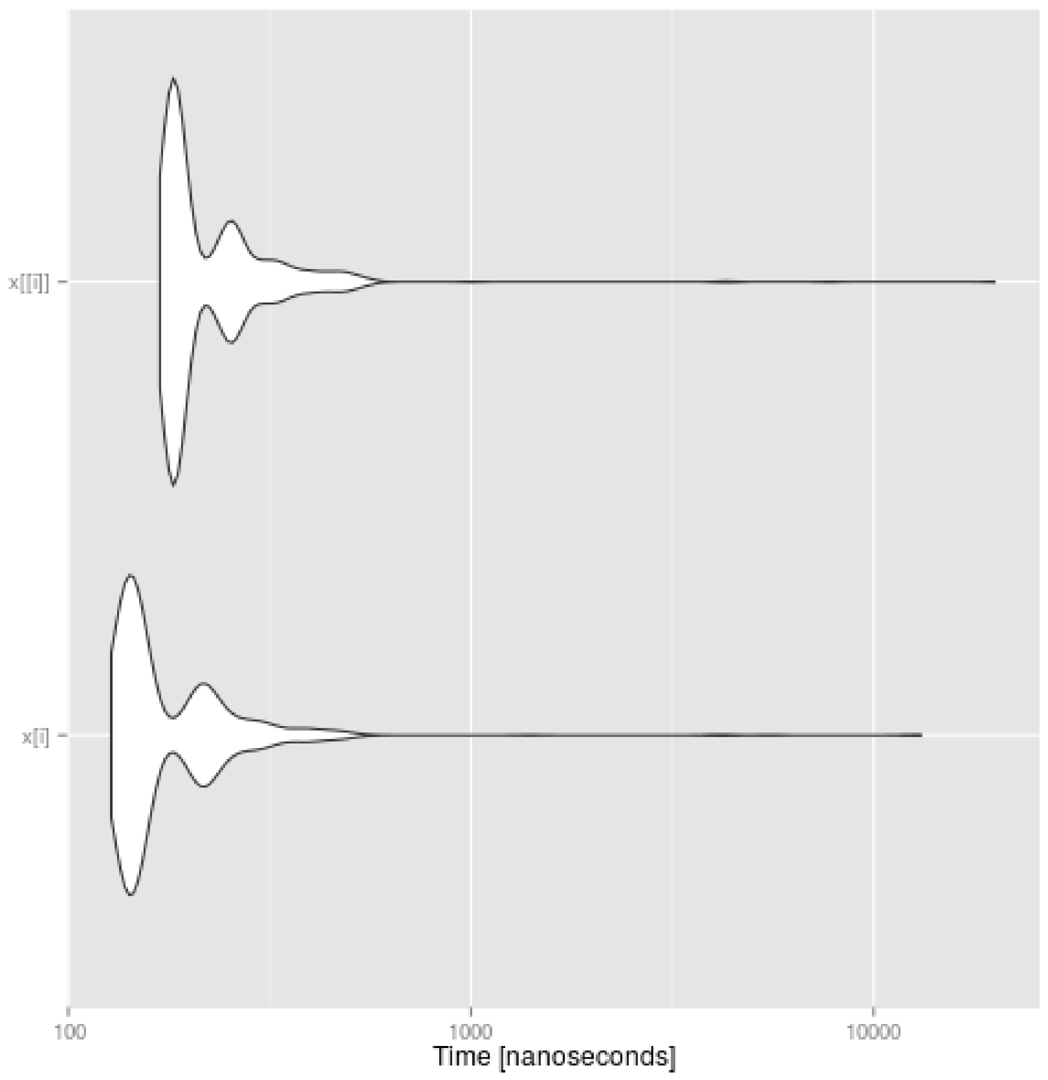


Figure 8.1: plot of chunk profiling-016

8.4 Poor man's debugging

Poor man's debugging doesn't require the knowledge of debug functions. Poor man's debugging is probably the simplest of the debugging methods, which often makes it the most effective!

It is an extremely easily implemented method which can tell you exactly where something went wrong.

It is based on `print()` and `cat()` functions. Differences between `print()` and `cat()` are described at the end of this section. This kind of debugging is done by simply printing some text between the lines, including the variables you are currently working with. For this reason, it is also called Debug by Print.

As a simple example, consider this basic function:

```
quick <- function(df) {
  plot(df$x, df$y, type = "b")
  summary(df)
}
```

This function works when `x` is numeric:

```
df1 <- data.frame(x = 1:100 , y = rnorm(100))
quick(df1)
```

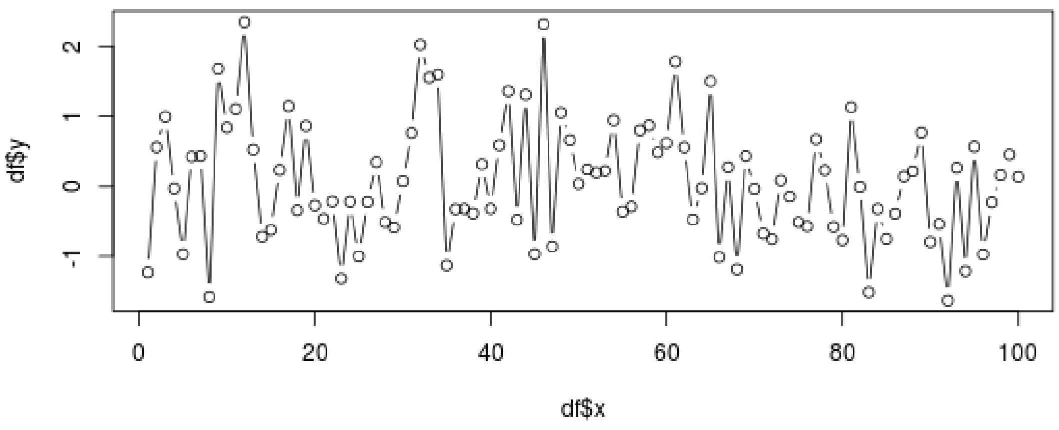


Figure 8.2: plot of chunk quick1

```
##        x               y
##  Min.   :  1.0   Min.   :-1.6295
##  1st Qu.: 25.8   1st Qu.:-0.5186
##   Median : 50.5   Median : 0.0569
##   Mean   : 50.5   Mean   : 0.0819
##   3rd Qu.: 75.2   3rd Qu.: 0.5954
##   Max.   :100.0   Max.   : 2.3499
```

The function works as well when `x` is a factor:

```
df2 <- data.frame(x = sample(letters[1:3], 100, rep= T ), y = rnorm(100))
quick(df2)
```

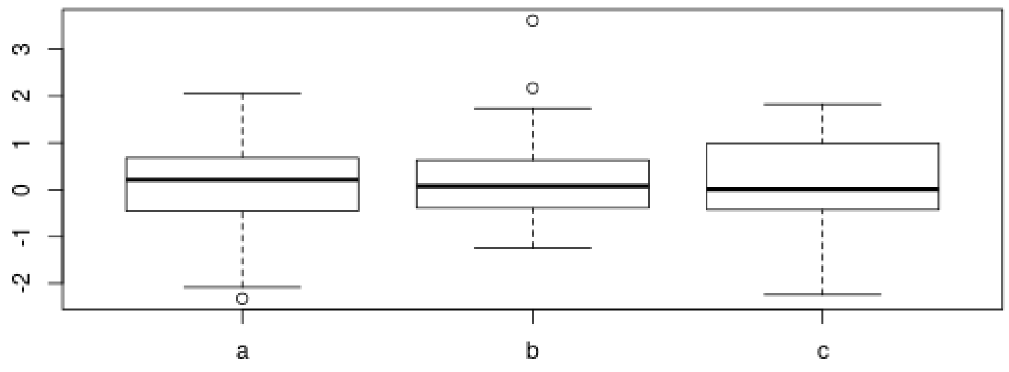


Figure 8.3: plot of chunk quick2

```
##   x             y
##   a:33   Min.   :-2.323
##   b:41   1st Qu.:-0.430
##   c:26   Median : 0.063
##          Mean   : 0.150
##          3rd Qu.: 0.762
##          Max.   : 3.614
```

The same function stops working when option `stringsAsFactors` is set to `FALSE`.

```
options(stringsAsFactors = FALSE)
df <- data.frame(x = sample(letters[1:3], 100, rep= T ), y = rnorm(100))
try(quick(df))
```

```
## Warning: NAs introduced by coercion
## Warning: no non-missing arguments to min; returning Inf
## Warning: no non-missing arguments to max; returning -Inf
```

As a very simple debugging procedure we could insert a `cat` command within the function body:

```
quick <- function(df){
 cat (str(df), "\n")
 plot(df$x, df$y, type = "b")
 summary(df)
}
quick(df)
```

```
## 'data.frame':    100 obs. of  2 variables:
##  $ x: chr  "a" "c" "c" "a" ...
##  $ y: num  -0.81 -0.676 0.068 -0.377 -2.101 ...
##
```

```
## Warning: NAs introduced by coercion
## Warning: no non-missing arguments to min; returning Inf
## Warning: no non-missing arguments to max; returning -Inf
```

```
## Error: need finite 'xlim' values
```

We could easily observe that `x` is a character vector and, as a consequence, it cannot be used as a plotting variable.

When using iterations, the use of markers may result in a very efficient debugging tool. `rmean()` is a trivial example illustrating the use of markers within a `for` loop.

```
rmean <- function(n, min, max){
 x <- numeric(n)
 for (i in 1:n){
   s <-  sample(min:max,1)
   x[i] <- log(s)
   if(!is.finite(x[i])) {cat ("Loop" , i, ": s = " , s , "\n")}
 }
 mean(x)
}

rmean(n = 10, min = -1, max = 4)
```

```
## Warning: NaNs produced
```

```
## Loop 2 : s =  -1
## Loop 7 : s =  0
```

```
## Warning: NaNs produced
```

```
## Loop 10 : s =  -1
## [1] NaN
```

This example fully illustrates how to take advantage of this debugging tecnique. In this instance, a marker `Loop` is used to indicate that the print out is within the loop and that `s` is not finite. This allows to quickly reference which part of the code is being processed.

The values of the variables that are being worked within the loop inside `rmean()` are returned. Now, if there is a simple flaw in our logic or something wrong was happening, this output makes most problems easy to track down.

In other words, when a function doesn't work may be useful to insert a `print()` or a `cat()` in several points of the function to detect where it fails.

8.4.1 print and cat

Both `print()` and `cat()` print a string, but several differences exist between these functions:

- `print()` is a generic method which looks at the class of its first argument and dispatches a different method depending on which class the first argument is. If no method is found, R uses `print.default`. Cat simply returns any object as it is. As an example we can consider any object belonging to a class with a print method related to that class:

```
xf <- factor(c("a", "b", "c"))
class(xf)

## [1] "factor"

print(xf)

## [1] a b c
## Levels: a b c

print.default(xf)

## [1] 1 2 3

cat(xf)

## 1 2 3
```

- `print()` requires `paste()` to concatenate strings, while `cat()` concatenates strings before outputting any result:

```
myCountry <- "Italy"
print(paste("I live in", myCountry))
```

```
## [1] "I live in Italy"
```

```
cat("I live in", myCountry)
```

```
## I live in Italy
```

- `cat()` interprets character strings that it gets.

```
xc <- 'test\\test'
print(xc)
```

```
## [1] "test\\test"
```

```
cat(xc)
```

```
## test\test
```

8.5 stop() and warning()

Functions `stop()` and `warning()` provide basic tools for exception handling. `stop()` stops execution of the current expression and executes an error action while `warning()` generates a warning message that corresponds to its argument. An extensive use of these functions during code development makes debugging much shorter.

```
half <- function(x) {return(x/2)}
try(half("text"))

half <- function(x){
  if(!is.numeric(x)) {stop("x must be numeric")}
  return(x/2)
}
half("text")
```

```
## Error: x must be numeric
```

```
repText <- function(times, text = NA) {
  if(is.na(text)) {
    warning("text not provided. 'test' is used.")
    text  <- "test"
  }
  rep(text, times)
}
repText(times = 3)
```

```
## Warning: text not provided. 'test' is used.
```

```
## [1] "test" "test" "test"
```

8.6 Trying a function

The function `try()` is a wrapper to run an expression that might fail and allows the user's code to handle error-recovery. It is useful to avoid that an error stops the execution of the whole function, as below. Again, good use of `try()` when developing avoid extensive debugging session.

```
doit <- function(x) {
  x <- sample(x, replace=TRUE)
  if(length(unique(x)) == length(x)) {
    mean(x)
  } else {
    stop("too few unique points")
  }
  cat("end of function", "\n")
  invisible(NULL)
}

x <- 1:10
thisError  <- doit(x)

## Error: too few unique points

thisTry <- try(doit(x))
thisError

## Error: object 'thisError' not found

thisTry

## [1] "Error in doit(x) : too few unique points\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in doit(x): too few unique points>
```

When function `doit()` is called directly, it returns an error and object `thisError` is not created. When using `try()` as a wrapper on `doit()`, despite the function returns error, object `thisTry` is created. `try()` returns the object returned by the called funcion if the called function does not go in error, otherwise, `try()` returns an object of class "try-error".

8.7 Debugging in R

There are a few basic tools in R that are worth knowing about:

- `browser()` interrupts the execution of an expression or within an R function. Objects can be viewed and changed during execution. Support exists for setting conditional breakpoints.
- `debug()` marks a function for debugging, so that `browser()` will be called on entry.
- `trace()` modifies a function to allow debug code to be temporarily inserted.

8.7.1 Browsing

Function `browser()` is usually called from within an other function. When the evaluator encounters this function it opens a browsing session on the evaluated frame so that the active environment can be explored.

All R commands can be used within the evaluated frame. Moreover, the interpreter, sets up special functions to be called uniquely within the `browser()` environment:

- `n` advances to the next step;
- `c` continues to the end of the current section of code (e.g., to the end of a for loop, or the end of the function itself);
- `where` prints a stack trace of all active function calls;
- `Q` exits the browser and the current evaluation and return to the top-level prompt.

As an example, consider the function below. It ouputs the sum of a given parameter plus two random numbers. In order to see the value of the two random numbers: `rn` and `ru` prior summing them up, the function evaluated frame is inspected by inserting a call to function `browser` within the function body.

```
mix <- function (x) {
 rn <- rnorm(1)
 ru <- runif(1)
 browser()
 return(x +rn+ru)
}

mix(0)
Called from: mix(0)
Browse[1]> rn
[1] 0.9315371
Browse[1]> ru
[1] 0.4610282
Browse[1]> c
[1] 1.392565
```

The use of `browser()` finds its best application when called by function `debug()`.

8.7.2 *Post Mortem* Debugging

Classical debugging is named *Post Mortem* as the error function is debugged after the evaluator returns a critical error so that the execution of the function is interrupted. Function `debug()` is the most used debugging tool.

Function `debug()` allows stepping through the execution of a function, line by line. At any step a `browser()` is opened on the evaluation frame.

As an example, consider this simple function:

```
msg <- function (x) {
  if (x > 0 ) cat ("Hello")
  else cat("goodbye")
  invisible(NULL)
}
```

```
msg(1)
```

```
## Hello
```

```
msg(-1)
```

```
## goodbye
```

This function simply prints *"Hello"* or *"Goodbye"* depending on whether `x` is greater than or less than 0. In principle, this function should never fail; nevertheless:

```
msg(log(-1))
```

```
## Warning: NaNs produced
```

```
## Error: missing value where TRUE/FALSE needed
```

this function returns an error and the execution is halted. As a first step, a call to `traceback()` is always worth:

```
traceback()
```

```
## No traceback available
```

In this case, `traceback()` does not help a lot. It simply says that the error occurs within the call to `msg()`. Not a big surprise as, `msg()` was the only function called.

The use of functions `debug()` or `debugonce()` returns interesting results:

```
debugonce(msg)
```

```
msg(log(-1))
debugging in: msg(log(-1))
debug at #1: {
    if (x > 0)
        cat("Hello")
    else cat("goodbye")
    invisible(NULL)
}
Browse[2]> ls()
```

```
[1] "x"
Browse[2]> x
[1] NaN
Warning message:
In log(-1) : NaNs produced
Browse[2]> x > 0
[1] NA
Browse[2]> n
debug at #2: if (x > 0) cat("Hello") else cat("goodbye")
Browse[2]> n
Error in if (x > 0) cat("Hello") else cat("goodbye") :
  missing value where TRUE/FALSE needed
```

As it can be observed, `log(-1)` produces a simple warning and returns `NaN` but, passing the resulting value of `x` into `msg()` ended in a fatal error as the comparison between `NaN` and `0` is clearly undefined and returns `NA` that, in tur, returns an error like the following:

```
if(NA) {cat ("this is strange")}

## Error: missing value where TRUE/FALSE needed
```

Finally, `debugonce()` is clearly used to debug a function at its next call. When using `debug()`, the function passed as an argument is set into "debug mode" and it is kept in the same mode until `undebug()` is called on the same function. Function `isdebugged()` is used to query the debugging flag on a function.

```
debug(msg)
isdebugged(msg)

## [1] TRUE

undebug(msg)
isdebugged(msg)

## [1] FALSE
```

In more complex situation, the use of `traceback()` after a call stack may result into a very useful debugging tool. Suppose we define functions `f()`, `g()` and `h()` as:

```
f <-  function(x) {
  r <-  x - g(x)
  r
}

g <- function(y) {
  r <- y * h(y)
  r
}

h <- function(z) {
  r <- log(z)
  if (r < 10)
      r^2
  else r^3
}
```

when calling the outer function `f()`:

```
f(-1)
```

```
## Warning: NaNs produced
```

```
## Error: missing value where TRUE/FALSE needed
```

The error seems to be generated within function `f()`. And this is partially true as the error is actually generated within function `h()` as `f()` calls `g()` that calls `h()`. All this mechanism is not always transparent to the end user as the inner mechanism of a function is not necessarily known. A simple call to traceback may quickly put the truth into evidence:

```
traceback()
```

```
3: h(y) at #2
2: g(x) at #2
1: f(-1)
```

The output of `traceback()` is to be read bottom-up or by following the output row number. In this case, `traceback()` says that the error occurs within function `h()`. As a result, function `h()` can be put into debug mode and `f(-1)` be called again:

```
debugonce(h)
f(-1)
```

```
debugging in: h(y)
debug at #1: {
    r <- log(z)
    if (r < 10)
        r^2
    else r^3
}
Browse[2]>
debug at #2: r <- log(z)
Browse[2]> ls()
```

```
[1] "z"
Browse[2]> z
[1] -1
Browse[2]> n
debug at #3: if (r < 10) r^2 else r^3
Browse[2]> ls()
[1] "r" "z"
Warning message:
In log(z) : NaNs produced
Browse[2]> z
[1] -1
Browse[2]> r
[1] NaN
Browse[2]> Q
```

The error clearly occurs in function `h()` as `NA` within an `if` statement returns error.

In case we simply put `f()` directly into debug mode instead of running `traceback()` first, the result would be useless:

```
debugonce(f)

f(-1)
debugging in: f(-1)
debug at #1: {
    r = x - g(x)
    r
}
Browse[2]>
debug at #2: r = x - g(x)
Browse[2]> ls()
[1] "x"
Browse[2]> x
[1] -1
Browse[2]> n
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced
```

8.7.3 Debugging on error

When the evaluator encounters an error, it looks for an error action. The error action, a function or an expression to be evaluated, is determined by the `error` argument of the `options`, by default set to `NULL`:

```
options("error")

## $error
## NULL
```

By setting:

```
options(error = recover)
```

the evaluator stops whenever an error occurs and calls function `recover()` which, in turn, offers a selection of available frames for browsing. By selecting 0, recovering is exited and the normal prompt is returned.

```
f(-1)
```

```
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced

Enter a frame number, or 0 to exit

1: f(-1)
2: #2: g(x)
3: #2: h(y)

Selection: 1
Called from: top level
Browse[1]> ls()
[1] "x"
Browse[1]> x
[1] -1
Browse[1]> n
Enter a frame number, or 0 to exit

1: f(-1)
2: #2: g(x)
3: #2: h(y)

Selection: 2
Called from: f(-1)
Browse[2]> ls()
[1] "y"
Browse[2]> y
[1] -1
Browse[2]> n

Enter a frame number, or 0 to exit

1: f(-1)
2: #2: g(x)
3: #2: h(y)

Selection: 3
```

```
Called from: g(x)
Browse[3]> ls()
[1] "r" "z"
Browse[3]> z
[1] -1
Browse[3]> r
[1] NaN

Browse[3]> h
function(z) {
  r <- log(z)
  if (r < 10)
      r^2
  else r^3
}

Browse[3]> r < 10
[1] NA

Browse[3]> #OK got it !!
Browse[3]> n

Enter a frame number, or 0 to exit

1: f(-1)
2: #2: g(x)
3: #2: h(y)

Selection: 0
```

Debugging on error is clearly a useful techniques when running R in interactive mode. When calling R in BATCH mode, interactive debugging is of no use. In this case a call to `dump.frames()` provides the right solutions as the evaluated frame is dumped into a file, `last.dump` by default, to be lately inspected by function `debugger()`. Again, `dump.frame()` is set as the error argument to function `options()`.

```
options(error = quote(dump.frames("fdump", to.file=TRUE)))
f(-1)

## Warning: NaNs produced

## Error: missing value where TRUE/FALSE needed
```

The error can now be inspected, as in interactive mode. The whole working environment has been dumped to file `fdump` that can be loaded, at any time, by function `debugger()`:

```
load("fdump.rda")
debugger(fdump)
```

```
Message:  Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
Available environments had calls:
1: f(-1)
2: #2: g(x)
3: #2: h(y)

Enter an environment number, or 0 to exit
Selection: 0
```

Finally, during development, setting `options("error")` as:

```
options(error = Quote(
 if (interactive()) recover()
 else dump.frames()
))
```

could help to take advantage of both solutions.

Chapter 9

Parallel computation

The R community has developed several packages to take advantage of parallelism.

Many of these packages are simply wrappers around one or multiple other parallelism packages forming a complex and sometimes confusing web of packages.

Package `parallel` attempts to eliminate some of this by wrapping `snow` and `multicore` into a nice bundle.

Parallel computation is especially suited to "embarrassingly parallel" problems like large-scale simulations and by-group analyses.

9.1 Package Parallel

Parallel computation can be divided into explicit and implicit parallelism.

9.1.1 Multicore parallelism

The `multicore` package, bundled within `parallel` provides a way of running parallel computations in `R` on machines with multiple cores or CPUs by making use of operating system facilities.

At present time, `multocore`, as based on process splitting, seems to work only on Unix like computers.

Function `mclapply`, works just like the regular `lapply` function to iterate across the elements of a list, but iterations automatically run in parallel to speed up the computations.

9.1.2 Example: : 2-dimensional function

We want to calculate the 2-dimensional `f()` function on a grid between `[-10, 10]` consisting of `1,000` points where `f()` is defined as:

```
f <- function(x) {
r <- sqrt(x[1]^2 + x[2]^2)
10 * sin(r) / r
}
```

We simulate some test data:

```
x <- seq(-10, 10, length.out=1000)
grid <- expand.grid(x=x, y=x)
head(grid, n = 3)
```

```
##        x   y
## 1 -10.00 -10
## 2  -9.98 -10
## 3  -9.96 -10
```

The sequential calculation simply calls function `f()` for each row of the grid:

```
system.time({z <- apply(grid, 1, f)})
```

```
##    user  system elapsed
##  14.964   0.147  15.129
```

The calculation needs about 20 seconds. To assure the correctness of the calculation we plot the function:

```
par(mfrow = c(1,1))
dim(z) <- c(length(x), length(x))
persp(x, x, z, theta=30, phi=30, expand=0.5, col='white', border=NA, shade=0.3, box=FALSE)
```

A strategy for a parallel computation of function `f()` is to split the grid into subgrids and calculate the function for these subgrids in parallel.

Here two cores are given, therefore the grid is split into two parts along `y = 0`:

```
grid_list <- split(grid, grid[,'y'] > 0)
```

Now, `apply()` from the sequential calculation is executed for each element of the list. A sequential execution is done using `lapply()`:

```
system.time({z_list <- lapply(grid_list, apply, 1, f)})
```

```
##    user  system elapsed
##   11.43    0.07   11.51
```

The parallel execution is done using the parallel analogon, `mclapply()`:

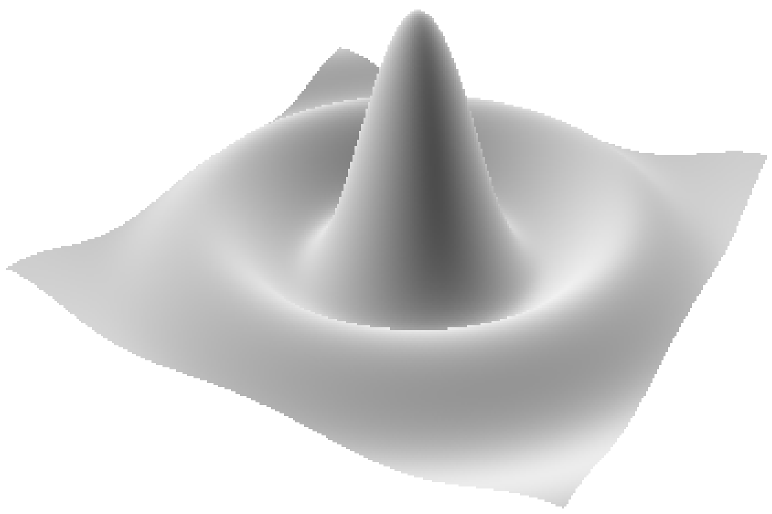


Figure 9.1: plot of chunk parallel-004

```
library(parallel)
system.time({z_list <- mclapply(grid_list, apply, 1, f, mc.cores = 2L, mc.preschedule=TRUE)})
```

```
##    user  system elapsed
##   5.118   0.196   5.297
```

The parallel execution needs about half of the time. In background the `mclapply()` has forked two child processes and each of the them has calculated the result for one list element. In principle, `mclapply()` produces the following results:

```
par(mfrow=c(1,2))
# First half:
x1 <- x[x > 0]
z1 <- z_list[[1]]
dim(z1) <- c(length(x), length(x1))
persp(x, x1, z1, theta=90, phi=30, expand=0.5, col='white',border=NA, shade=0.3, box=FALSE)

# Second half:
x2 <- x[x <= 0]
z2 <- z_list[[2]]
dim(z2) <- c(length(x), length(x2))
persp(x, x2, z2, theta=90, phi=30, expand=0.5, col='white', border=NA, shade=0.3, box=FALSE)
```

For the final result the elements of the list must be put together:

```
z <- Reduce(c, z_list)
```

9.1.3 Example: Cross Validation

We can define a simple cross validation function for `lm` type objects as follows:

```
cross_val <- function(i, fm){
  data <- fm$model
  formula <- formula(fm)
  response <- as.character(terms(fm)[[2]])
  fm <- lm(formula, data = data[-i,])
  newdata <- data[i,]
  predicted <- predict (fm, newdata)
  sqrt((predicted - data[i,response] )^2)
}
```

We create a `1,000` rows dataframe and a linear regression model applied to it.

```
n <-5*10^3
df <-  data.frame(x = 1:n,y = 2+3*c(1:n)+rnorm(n, 0, 1500))
df$y[800] <-  2*10^4
plot(y~x , data = df, pch = 16, cex = .5)
```

Figure 9.2: plot of chunk parallel-008

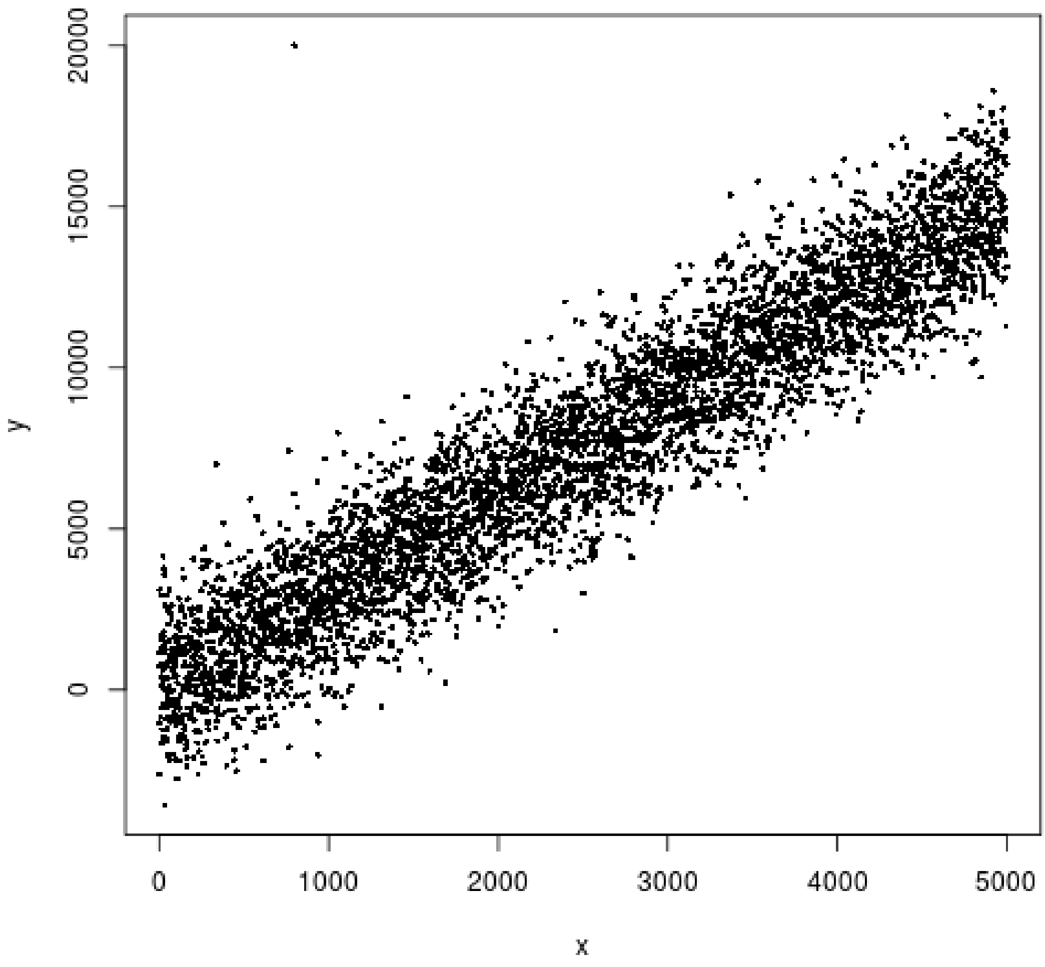


Figure 9.3: plot of chunk dataframeDf

```
fm <-  lm(y~x, data = df)
```

The previously created `cross_val()` function can be iterated trough `lapply()` using a single core:

```
system.time({single <- lapply(1:n, cross_val, fm)})[3]
```

```
## elapsed
##   30.79
```

```
op <- par(mfrow = c(1, 2))
plot(y~x , df, pch = 16, col = "red", cex = .6)
plot(unlist(single), type = "s", ylab = "Cross Validation", col = "darkgreen")
```

```
par(op)
```

By using `mclapply()`, parallelism is achieved straightforwardly without the need of setting an explicit cluster environment:

```
system.time({
  quad <- mclapply(1:n, cross_val, fm,
  mc.cores = 4L, mc.preschedule = TRUE)})[3]
```

```
## elapsed
##   10.95
```

Clearly, both methods lead to the same results:

```
identical(single, quad)
```

```
## [1] TRUE
```

9.1.4 Summary statistics revisited

We can revisit the previous example about a generic summary function:

```
my_summary <- function(x, flist){
  f <- function(f,...)f(...)
  g <- function(x, flist){vapply(flist, f , x, FUN.VALUE = numeric(1))}
  df <- as.data.frame(lapply(x, g , flist))
  row.names(df) <- names(flist)
  df
}
```

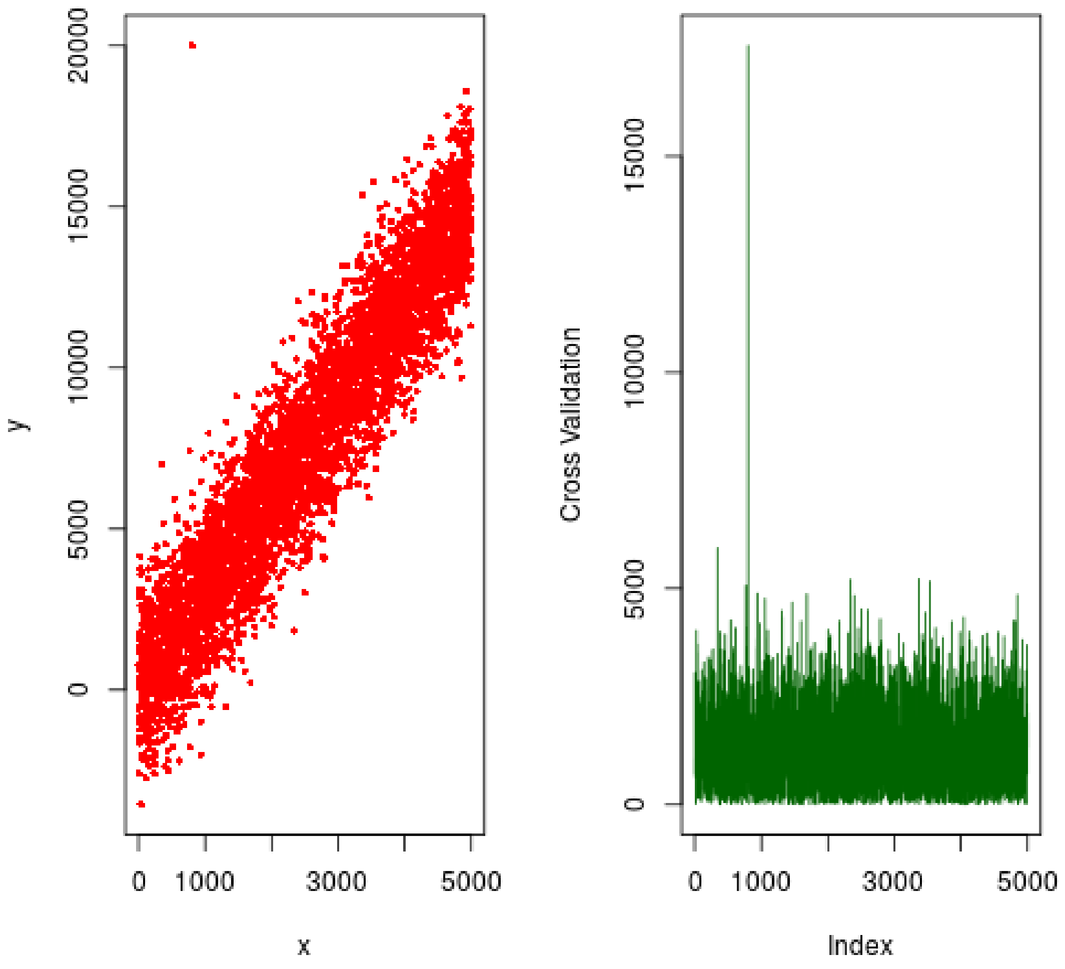


Figure 9.4: plot of chunk crossValsingleCore

Suppose we have a dataframe of about `0.4 Gb`

```
n <- 10^7
df <- data.frame(x1 = rnorm(n, 10, 1), x2 = rweibull(n, 1, 2), x3 = rpois(n, 100),
x4 = rnorm(n, 10, 1), x5 = rweibull(n, 1, 2), x6 = rpois(n, 100))
print(object.size(df), units = "Gb")
```

```
## 0.4 Gb
```

We may use `my_summary()`

```
system.time({
my_summary(df,
  flist = list(mean = mean, stdev = sd, range =  function(x,...){sd(x,...)/mean(x,...)})
)})
```

```
##    user  system elapsed
##   1.788   0.066   1.858
```

or write a parallel version of this function with minor modifications to teh original `my_summary()`:

```
my_mc_summary <- function(x, flist, mc.cores = 1L){
  f <- function(f,...)f(...)
  g <- function(x, flist){vapply(flist, f , x, FUN.VALUE = numeric(1))}
  df <- as.data.frame(mclapply(x, g , flist, mc.cores = mc.cores))
  row.names(df) <- names(flist)
  df
}
```

and use it as

```
system.time({
my_mc_summary(df,
  flist = list(mean = mean, stdev = sd, range =  function(x,...){sd(x,...)/mean(x,...)}),
  mc.cores = 2L
)})
```

```
##    user  system elapsed
##   0.555   0.082   0.684
```

The reduction in computing time is appreciabl eat very little coding effort.

Finally, `my_mc_summary(..., mc.cores = 1L)` will run the `mc` function on a single core as `my_summary()`.

9.1.5 Prescheduling

argument `mc.preschedule()` of `mclapply()` controls how data are allocated to processes and is set to `TRUE` by default.

If `mc.preschedule` is `TRUE`, then the data is divided into `n` sections a priori and passed to `mc.cores` processes.

If `mc.preschedule` is `FALSE`, then a job is constructed for each data value sequentially, up to `mc.cores` at a time.

`mc.preschedule` set to `TRUE` is better for short computations or large number of values in `X` while `mc.preschedule` set to `FALSE` is better for jobs that have high variance of completion time and not too many values of `X`.

A convolution is good example where `mc.preschedule` surely needs to be set at `TRUE`. We need to pass `X = n = 10^4` data to functional `mclapply()`.

```
# mc.preschedule = TRUE
lambda <- 1000
n <- 10^4

f <- function(i, np, lambda, meanlog, sdlog){
  sum(rlnorm(np[i], meanlog, sdlog))
}

# mc.preschedule = FALSE
system.time({
  np <- rpois(n, lambda)
  out <- mclapply(X = 1:n, FUN = f,
    np = np, meanlog = 9,  sdlog = 2,
    mc.preschedule = FALSE)
})

##    user  system elapsed
##   19.02  207.41  107.58
```

Explicit parallelism has the programmer responsible for dividing the problem to be solved into independent chunks, to be run in parallel, and also responsible for aggregating the results from each chunk.

9.1.6 Cluster parallelism

While `maclaplly()` works on the basis of process forking abstactiong the user for the need of managing the underneath parallel backend, `R` offers other functionality for creating a set of copies of R running in parallel and communicating over sockets.

This method require the user to set up the cluster prior using it but, on the other hand, allows parallel computation to be extended over several machines.

We can asily define `nc` “workers” on a single computer by calling `makeCluster(nc)`:

```
library(parallel)
nc <- detectCores()
cluster <- makeCluster(nc)
cbind(clusterCall(cluster, function() {Sys.info()["nodename"]}))
```

```
##      [,1]
## [1,] "mutolo"
## [2,] "mutolo"
## [3,] "mutolo"
## [4,] "mutolo"
```

```
stopCluster(cluster)
```

`clusterCall()` calls a function on each node, whereas `stopCluster(cluster)` closes the cluster parallel environment previously created.

Notice that actually, when a parallel computation environment is created with `makeCluster()`, a "master" process with a number of *workers* or *slaves* processes are run. The role of master process is to *manage* worker processes and *join* the results, while slave processes actually perform the computations.

Then, after `makeCluster(nc)` call in previous script, `nc + 1` R processes shall run: a master process, and `nc` workers processes.

```
fx <- function(x){x+1}
cluster <- makeCluster(2)
clusterCall(cluster, fx, x = 5)
```

```
## [[1]]
## [1] 6
##
## [[2]]
## [1] 6
```

```
stopCluster(cluster)
```

Variables defined at master level are not directly available to all slaves. Therefore, if this example works

```
fxx <- function(x){x + xx}
xx <- 3
fxx(1)
```

```
## [1] 4
```

the following will fail

```
cluster <-  makeCluster(2)
clusterCall(cluster, fxx, x = 2)
```

```
## Error: 2 nodes produced errors; first error: object 'xx' not found
```

```
stopCluster(cluster)
```

Whenever required we'll have to export master variables to all slaves

```
xx <-  1
cluster = makeCluster(2)
clusterExport(cluster, "xx")
clusterCall(cluster, fxx, x = 2)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 3
```

```
stopCluster(cluster)
```

Similarly, we have to attach or required libraries at slave level. The function `clusterEvalQ()`, as it evaluates an expression at each cluster node, is the ideal candidate to achieve this task:

```
cluster <-  makeCluster(2)
clusterEvalQ(cluster, library(MASS))
```

```
## [[1]]
## [1] "MASS"      "methods"   "stats"     "graphics"  "grDevices" "utils"
## [7] "datasets"  "base"
##
## [[2]]
## [1] "MASS"      "methods"   "stats"     "graphics"  "grDevices" "utils"
## [7] "datasets"  "base"
```

```
stopCluster(cluster)
```

When calling `makeCluster()`, if no type argument is supplied, it defaults to `type = "PSOCK"` that calls `makePSOCKcluster`.

`makePSOCKcluster` is very similar to `makeSOCKcluster` in package `snow`. It runs `Rscript` on the specified host(s) to set up a worker process which listens on a socket for expressions to evaluate, and returns the results.

A simple convolution based on `lapply()` represents a good example for cluster parallelism. We can test the the *standalone* convolution by:

```
lambda <- 1000
n <- 10^5

f <- function(i, np, lambda, meanlog, sdlog){
  sum(rlnorm(np[i], meanlog, sdlog))
}

system.time({
  np <- rpois(n, lambda)
  out <- sapply(X = 1:n, FUN = f,
    np = np, meanlog = 9,  sdlog = 2)
  cat ("95th quantile = " , quantile(out , .95), "\n")
})

## 95th quantile =  81480802

##    user  system elapsed
##  13.310   0.091  13.414
```

We just need to replace the `sapply` function with the corresponding `parSapply` to make use of `R` parallel capabilities.

```
nc <- detectCores()
cluster  <- makeCluster(nc)

f <- function(i, np, lambda, meanlog, sdlog){
  sum(rlnorm(np[i], meanlog, sdlog))
}

system.time({
  np <- rpois(n, lambda)
  out <- parSapply(cl = cluster , X = 1:n, FUN = f,
    np = np, meanlog = 9,  sdlog = 2)
  cat ("95th quantile = " , quantile(out , .95), "\n")
})

## 95th quantile =  81401522

##    user  system elapsed
##   0.109   0.005   4.462

stopCluster(cluster)
```

9.1.7 Setting up a cluster

To run the same procedure in parallel on several machines, a proper network environment must be set up with public keys and hosts files.

The following procedure has to be executed only once in order to configure the network environment and set up all necessary permissions.

1. All computers require `ssh server` and `ssh-client` installed
2. Master being able of comunicatng with slaves via `ssh keys`
3. Master being able of comunicatng with itself via `ssh keys`
4. Master being able to comunicate to slaves over a given port in the range 11000:11999
5. Hosts names are properly setup in `/etc/hosts`

Note that we can generate a `ssh key` with `ssh-keygen -t rsa` and copy the `key` to the remote machine by `ssh-copy-id user@remote`

Unfortunately, when creating a snow (or parallel) cluster object many things can go wrong, and the most common failure mode is to hang indefinitely. In addition to `ssh` issues, the problem could be:

- `R` not installed on a worker machine
- `snow` not installed on a the worker machine
- `R` or `snow` not installed in the same location as the local machine
- current user doesn't exist on a worker machine
- networking problem
- firewall problem

and there are no doubt more possibilities.

In our experience, the single most useful troubleshooting technique is manual mode. Just set "manual" to TRUE when creating the cluster object. It's also a good idea to set "outfile" to the empty string so that you're more likely to see useful error messages:

```
cl <- makeSOCKcluster(ip_slave, manual=TRUE, outfile="")
```

`makeSOCKcluster()` will display an Rscript command to execute in a terminal on the specified machine. Obviously, this bypasses any ssh issues, and you will quickly learn if `R` or `snow` is not installed in the expected location. If we are lucky, we will get an error message and that will lead us to the solution

9.1.8 Example: Cluster convolution

```
ip_master <- "localhost"
ip_slave <- "localhost"

biCluster  <- makeCluster(spec = c(rep(ip_master, 2) , rep(ip_slave, 2)), port = 11001)
cbind(clusterCall(biCluster, function() {Sys.info()["nodename"]}))

##      [,1]
## [1,] "mutolo"
## [2,] "mutolo"
## [3,] "mutolo"
## [4,] "mutolo"
```

```
stopCluster(biCluster)
```

As the results show, the above instructions created a parallel computational environment with four slave processes at `master` and four slave processes `slave`

Now the convolution exercise can be easily divided among eight cores on two networked machines.

```
biCluster <- makeCluster(spec = c(rep(ip_master, 2), rep(ip_slave, 2)))

lambda <- 1000
n <- 10^6

f <- function(i, np, lambda, meanlog, sdlog) {
    sum(rlnorm(np[i], meanlog, sdlog))
}

system.time({
    np <- rpois(n, lambda)
    out <- parSapply(cl = biCluster, X = 1:n, FUN = f, np = np, meanlog = 9,
        sdlog = 2)
    cat("95th quantile = ", quantile(out, 0.95), "\n")
})

## 95th quantile =  81504374

##    user  system elapsed
##   1.145   0.036  38.850

stopCluster(biCluster)
```

Finally, working across public networks, especially with little bandwidth, may easily kill the extra benefits of having multiple cores available

9.2 Package Foreach

When loading the `foreach` packages, R displays:

```
library(foreach)
foreach: simple, scalable parallel programming from Revolution Analytics
Use Revolution R for scalability, fault tolerance and more.
http://www.revolutionanalytics.com
```

And in fact `foreach` is a `R` library developed at **Revolution Analytics**: the major commercial supplier of R based solutions.

Package `foreach` provides a new looping construct for executing R code repeatedly.

```
library(foreach)

## Error: there is no package called 'foreach'

foreach ( i=1:3) %do% log(i)

## Error: could not find function "%do%"
```

Function `foreach()` is similar to the standard function `lapply`, but doesn't require the evaluation of a function. Moreover, `foreach()` has some interesting functionality such as the combine argument that makes `foreach` a very versatile iterator.

Argument `.combine` , combines results after the execution of the loop:

```
foreach(i = 1:3, .combine = cbind) %do% sample(1:5, 5)

## Error: could not find function "%do%"
```

In the case, `foreach` outputs a matrix as function cbind returns.

The main reason for using package `foreach` is that it supports parallel execution, that is, it can execute those repeated operations on multiple cores, or on multiple nodes of a cluster.

Changing from the single core to the multi core version of a foreach loop results in a quite a simple task:

```
library(doMC)

## Error: there is no package called 'doMC'

registerDoMC(cores = 2)

## Error: could not find function "registerDoMC"

result <- foreach ( i=1:100) %dopar% log(i)

## Error: could not find function "%dopar%"
```

Clearly, with this example, no advantage exists when using the parallel version of the foreach iterator. This is is only for teaching purposes.

Note that foreach needs a parallel backend to be able to run in `%dopar%` mode. The parallel backend, in this case, is provided by function `registerDoMC()`. This function is used to register the multicore parallel backend with the foreach package.

Chapter 10

‘Rcpp‘: Seamless ‘R‘ and ‘C++‘ Integration

10.1 Introduction

Sometimes R code just isn’t fast enough and there’s no simply way to make the code any faster.

The Rcpp package provides a consistent API for seamlessly accessing, extending or modifying R objects at the C++ level. The API is a rewritten and extented version of an earlier API which we refer to as the ‘classic Rcpp API’.

10.2 Rcpp, R and C++

All examples in this chapter need at least version 0.10.1 of the Rcpp package. This version includes cppFunction and sourceCpp, which makes it very easy to connect C++ to R. You’ll also need a working C++ compiler.

Before using any of Rccp facilities, we need at least:

```
require(Rcpp)
```

```
## Loading required package: Rcpp
```

Moreover, in order to perfom some comparisons, we also load:

```
require(rbenchmark)
```

```
## Loading required package: rbenchmark
```

```
require(microbenchmark)
```

```
## Loading required package: microbenchmark
```

10.3 A simple example with scalars

`cppFunction` allows developer to write C++ functions in R like this:

```
cppFunction('
  int add(int x, int y, int z) {
    int sum = x + y + z;
    return sum;
  }'
)
```

Once the function is defined, its print method displays info about the function

```
add
```

```
## function (x, y, z)
## .Primitive(".Call")(<pointer: 0x2b7921338c10>, x, y, z)
```

and we can use it as any other `R` function:

```
add(1, 2, 3)
```

```
## [1] 6
```

Some notes about R and C++ that may be worth to recall before moving forward might be:

- In C++ we must declare the type of output the function returns.
- The classes for the most common types of R vectors are: `NumericVector`, `IntegerVector`, `CharacterVector` and `LogicalVector`.
- Scalars and vectors in C++ are different.
- The scalar equivalents of `numeric`, `integer`, `character` and `logical` `R` vectors are, respectively, `double`, `int`, `String` and `bool` in `C++`.

10.4 Brief introduction to vectors

Now let us try a slightly more complex example.

```
cppFunction('
  double sumC(NumericVector x) {
    int n = x.size();
    double total = 0;
    for(int i = 0; i < n; ++i) {
      total += x[i];
    }
    return total;
  }
')
```

The above function simply returns the sum of elements of a vector passed as a parameter to function `sumC()`.

Note the use of `NumericVector` to "say" to `sumC()` that the argument `x` is a vector containing floating point (double precision) numeric values.

Note also that the C++ code performs an explicit loop on all the vector elements to calculate the sum.

`sumC()` function is available in `.Global` environment, as all R functions created by user:

```
ls()
```

```
## [1] "add"  "sumC"
```

And the speed of `sumC()` is comparable to speed of native R `sum()` function:

```
y=rnorm(1000000)
microbenchmark(
  sumC(y),
  sum(y)
)
```

```
## Unit: milliseconds
##     expr   min    lq median    uq   max neval
##  sumC(y) 1.002 1.012  1.036 1.057 1.198   100
##   sum(y) 1.021 1.047  1.055 1.074 1.201   100
```

Now let us try to produce a yet more complex example: a function that calculates the inner product between two equal-length vectors:

```
cppFunction('
  double sumP(NumericVector x, NumericVector y) {
    int nx = x.size();
    int ny = y.size();
    double sumup=0;

    if(nx==ny)
    {
      for(int i=0; i < nx; i++)
          sumup += x[i] * y[i];
    }
    else
        sumup = NA_REAL; // NA_REAL: constant of NA value for numeric (double) values

    return sumup;
    }'
)
```

We than compare our newly created function with two equivalent `R` native approaches:

```
microbenchmark(
  sumP(rnorm(1E5),rnorm(1E5)),
  sum(rnorm(1E5)*rnorm(1E5)),
  t(rnorm(1E5)) %*% rnorm(1E5)
)
```

```
## Unit: milliseconds
##                              expr   min    lq median    uq   max neval
##   sumP(rnorm(1e+05), rnorm(1e+05)) 16.19 16.66  16.78 17.18 17.97   100
##   sum(rnorm(1e+05) * rnorm(1e+05)) 16.35 16.64  16.86 17.17 44.39   100
##  t(rnorm(1e+05)) %*% rnorm(1e+05) 17.22 17.51  17.72 18.24 46.14   100
```

In above example, the function `sumP()` seems a just about faster than both `sum()` and the linear algebra-based calculation. Moreover, some small changes to the function may reveal quite sensible advantages either in terms of memory usage, or in terms of performance:

```
x <- rnorm(1E8)
y <- rnorm(1E8)

#Test C function
mmC1=gc(reset=TRUE)
tmC=system.time(sumP(x,y))
mmC2=gc()

#Test standard R
mmR1=gc(reset=TRUE)
tmR=system.time(sum(x*y))
mmR2=gc()

#Test Matrix calculation
mmR3=gc(reset=TRUE)
tmR1=system.time(t(x) %*% y)
mmR4=gc()
```

The amount of memory requested by `sumP()` to perform the computations is close to 0: `mmC2[2,6]-mmC1[2,6] =` 0.1 MB, whereas `sum()` needed `mmR2[2,6]-mmR1[2,6] =` 763 MB.

This because the calculation performed by `sum()` is made of two steps:

1. Calculate the vector containing the element by element product
2. Sum the values of products

This produced another side effct: the time requested to perform the inner product alone with `sumP()` is about 40% of the time requested by `sum()`, since some time was needed to create the vector and the two calculations “steps” (instead of only one of `sumP()`) in `sum()`.

The same calculations performed by linear algebra operators return the worst performances both in terms of execution speed and in terms of memory usage.

That is, by a simple function written in C++, we gain sensibily faster and less memory hungry code.

Finally, we could object that the `crossprod()` R function does the same computations performed by `sumP()`. This is true, but as an example, this could be equally useful.

Now, let's try a new problem: we need a function that "generates" ("simulates") an `n`-length realization from an AR(1) process with parameter `phi`, starting from `start` and having a white noise error with standard deviation equal to `sigma`.

In this case the return value is not a scalar but a vector. However, the solution is simple:

```
cppFunction('
  NumericVector arC(double start, int n, double phi, double sigma) {
    RNGScope scope;
    NumericVector out(n);

    out[0]=start;
    for(int i = 1; i < n; i++) {
      out[i] = phi*out[i-1]+rnorm(1,0,sigma)[0];
    }
    return out;
  }
')
```

The `arC()` function will return a `n`-length vector directly manageable by R.
Notice in above code the use of `rnorm()` function, that behave exactly as the R `rnorm()` function. In Rcpp *sugar* (an abstraction layer of Rcpp that provide some facilities to the developer) all the main r/p/q/d R "vector" distribution functions are directly available to programmer.
Notice also the `RNGScope scope;` line of code, which sets the random number generatator on entry to a block and resets it on exit.

Let's try the function:

```
set.seed(1000)
phi=.7
ts=arC(10,500,phi,2)
plot(ts,type="l",
     main=paste("Plot of AR(1) Rcpp realization with parameter",phi))
abline(h=0,col="blue",lty=2)
grid()


op=par(mfrow=c(2,1))
acf(x=ts)
pacf(x=ts)
```

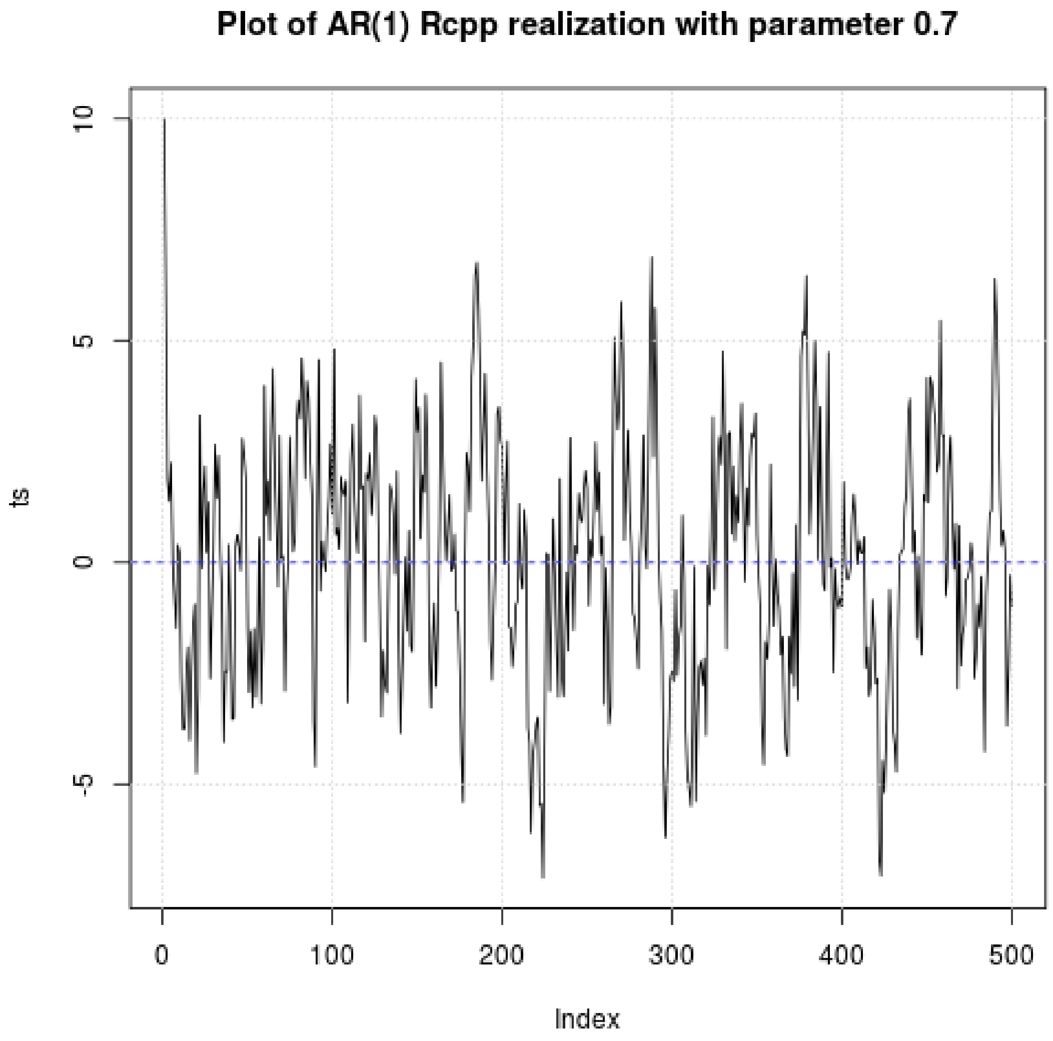


Figure 10.1: Realization of an AR(1) process with Rcpp

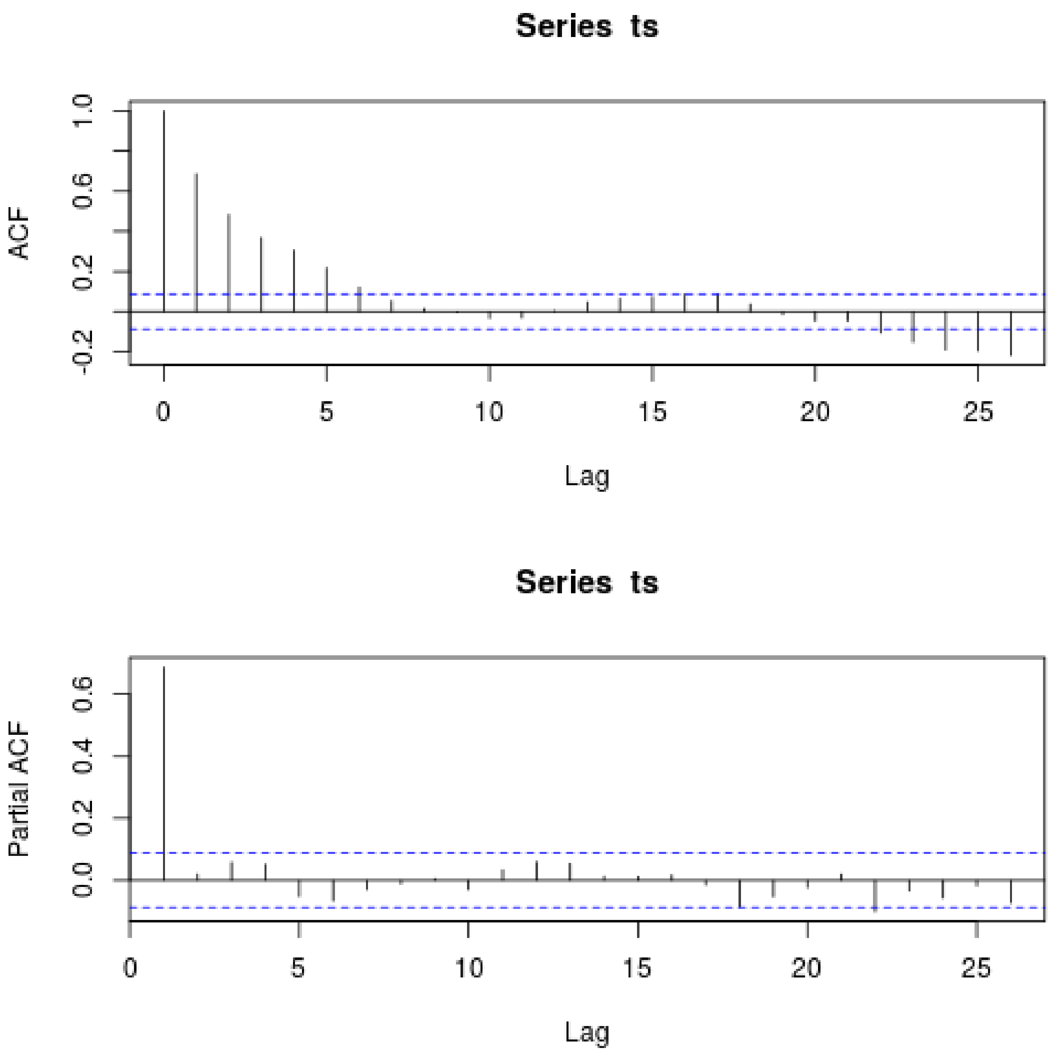


Figure 10.2: ACF and PACF of AR(1) process instance made with Rcpp

```
par(op)
```

Obviously, to avoid many calls to `rnorm()` in above function, a unique call like `rnorm(n,0,sigma)` can be made, but this requires more RAM:

```
cppFunction('
  NumericVector arC2(double start, int n, double phi, double sigma) {
    RNGScope scope;
    NumericVector out(n);
    NumericVector rn=rnorm(n,0,sigma);

    out[0]=start;
    for(int i = 1; i < n; i++) {
      out[i] = phi*out[i-1]+rn[i];
    }
    return out;
  }
')
```

Assuming, we want to write a similar function in `R`, we must write more complicated or much slower code. The simplest, and perhaps slowest, solution is:

```
arR=function(start, n , phi, sigma){
  out=numeric(n)
  out[1]=start

  for( i in 2:n){
    out[i]=phi*out[i-1]+rnorm(1,sd=sigma)
  }
  return(out)
}
```

And this is the plot:

```
set.seed(1000)
phi=.7
tsR=arR(10,500,phi,2)
plot(ts,type="l",
     main=paste("Plot of AR(1) R realization with parameter",phi))
abline(h=0,col="blue",lty=2)
grid()
```

Note that, given the assigned seed, the Rcpp and R functions produce exactly the same numeric output. That means that the compiled C++ code produced by Rcpp is totally integrated with R libraries:

```
sum(abs(ts-tsR))
```

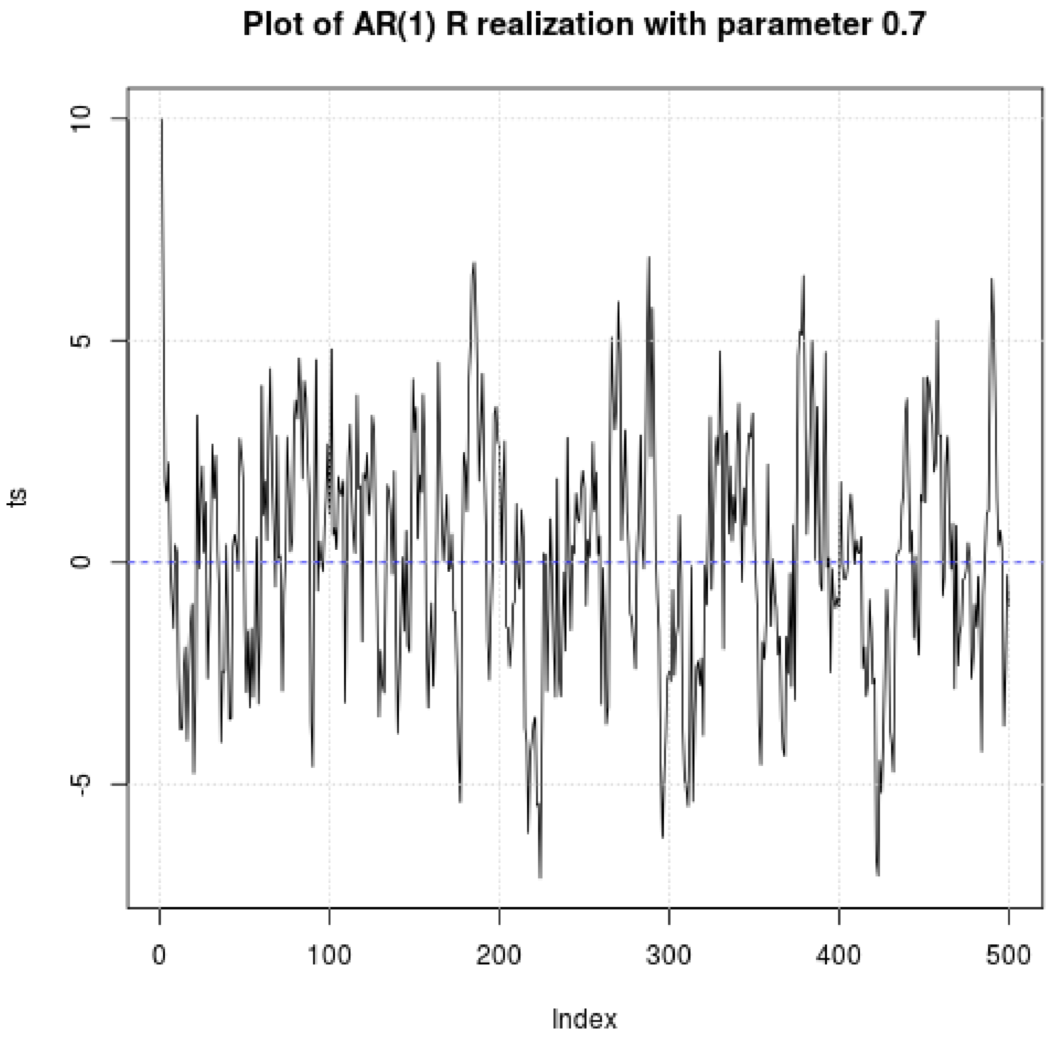


Figure 10.3: Realization of an AR(1) process with R

```
## [1] 0
```

A slightly better R implementation of the same function could be:

```
arR2=function(start, n , phi, sigma){
  out=numeric(n)
  out[1]=start
  err=rnorm(n,sd=sigma)

  for( i in 2:n){
    out[i]=phi*out[i-1]+err[i]
  }
  return(out)
}
```

The four above functions give the following performance results:

```
(mb=microbenchmark(
  arC(10,500,phi,2),
  arC2(10,500,phi,2),
  arR(10,500,phi,2),
  arR2(10,500,phi,2)
))
```

```
## Unit: microseconds
##                   expr     min      lq  median      uq     max neval
##    arC(10, 500, phi, 2)   50.39   54.11   56.86   59.74   71.03   100
##   arC2(10, 500, phi, 2)   33.12   35.13   36.51   38.93   59.90   100
##    arR(10, 500, phi, 2) 1940.79 2002.79 2029.68 2060.74 3306.61   100
##   arR2(10, 500, phi, 2)  625.46  651.32  665.47  694.64 2106.98   100
```

with average performances from 57 to 13 times better for C++ code.

10.5 Using sourceCpp

To simplify the initial presentation, the examples in this chapter have used inline C++ via `cppFunction()`. For real problems, it’s usually easier to use standalone C++ files and then source them into R using the `sourceCpp()` function.
This will enable us to take advantage of text editor support for C++ files (e.g. syntax highlighting) as well as make it easier to identify the line numbers of compilation errors.
Standalone C++ files can also contain embedded R code in special C++ comment blocks. This is really convenient if you want to run some R test code.

The standalone C++ file should have extension .cpp, and needs to start with:

```
#include <Rcpp.h>
using namespace Rcpp;
```

And for each function that we want available within `R`, we need to prefix it with:

```
// [[Rcpp::export]]
```

where the space after `//` is mandatory

Then using `sourceCpp("path/to/file.cpp")` will compile the C++ code, create the matching R functions and add them to current session.
Note that these functions, and the ones created using `cppFunction()`, will not persist across `save()` and `load()`, such as when you restore your workspace.

Let's try to copy/paste following code into a `cppExample1.cpp` file in current directory, and compile/run the script as below:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double meanC(NumericVector x) {
  int n = x.size();
  double total = 0;

  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total / n;
}

/*** R
  library(microbenchmark)
  x <- runif(1e5)
  microbenchmark(
    mean(x),
    meanC(x))
*/
```

Note the R code block embedded between `/*** R` and `*/`.
This code will be executed after the compilation to test the functions.

```
sourceCpp("cppExample1.cpp")
```

```
##
## > library(microbenchmark)
##
## > x <- runif(1e+05)
##
## > microbenchmark(mean(x), meanC(x))
## Unit: microseconds
##      expr    min     lq median     uq   max neval
##   mean(x) 189.34 190.27 195.52 197.23 254.4   100
##  meanC(x)  93.32  93.98  96.38  97.18 119.4   100
```

And now `meanC()` is available in current workspace:

```
"meanC" %in% ls()
```

```
## [1] TRUE
```

```
meanC
```

```
## function (x)
## .Primitive(".Call")(<pointer: 0x2b79220f2c50>, x)
```

10.6 Matrix computations example

Now we want to write a very simple function that calculates a correlation matrix from a data matrix.
For sake of simplicity, we suppose that the input matrix is numeric and that it does not contain NAs.

In Rcpp, the data types for integer, double, logical, and character matrices are `IntegerMatrix`, `NumericMatrix`, `LogicalMatrix`, `CharacterMatrix`, respectively.

Of course, the return value of our example function shall be an object of `NumericMatrix` type.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix corrC(NumericMatrix X) {

  Environment stats("package:stats");
  Function corr=stats["cor"];
  NumericMatrix out=corr(X);

  return out;
}
```

The above code is saved in `cppExample2.cpp` source C++ file.
Notice that the R `cor()` function of `stats` package has been used within the code. To use the function, an object of class `Environment` “containing” the `stats` package has been created, and then from this object a class `Function` object named `corr` has been created, where the `cor()` function of `stats` package has been assigned.
With these two simple lines of code, the R `cor()` function has became available to C++ code.
Now we can launch the compilation of code:

```
sourceCpp("cppExample2.cpp")
```

And now we can compare the performances of R and Rcpp implementations:

```
X=matrix(rnorm(100000),nrow=1000,ncol=100)
microbenchmark(
  corrC(X),
  as.matrix(cor(X))
)
```

```
## Unit: milliseconds
##               expr   min    lq median    uq   max neval
##           corrC(X) 5.214 5.281  5.364 5.425 5.809   100
##  as.matrix(cor(X)) 5.193 5.271  5.344 5.406 5.853   100
```

In this specific case, the differences are very small, since the two codes actually implement the same function.

10.7 A last more complex example

The `truncgof` library contains functions that allow one to calculate GoF statistics, and related p-values, for truncated distributions. The main drawback of these functions is that they use Montecarlo simulations to calculate p-values.
Since the estimates are often obtained by maximization of likelihood, the p-value calculation become a very time-consuming exercize, since for left-truncated distributions, as in this case, the maximum of likelihood is seldom obtainable in closed form, and then it requires a numeric optimization as provided by `optim()` or similar functions.

We would then like to speed-up the optimization process, and then the p-value calculation too.

Let us try an example: suppose that we have to estimate via MLE the parameters of a left-truncated lognormal distribution. The left truncated distribution function is:

```
dtrlnorm=function(x,pars,L=0,log=FALSE){
  if(L<=0){
    L=0
  }
  if(log){
    ret=log(x>=L) +
      dlnorm(x,meanlog=pars[1],sdlog=pars[2],log=log)-
      log(1-plnorm(L,meanlog=pars[1],sdlog=pars[2]))
  }
  else{
    ret=(x>=L)*
      dlnorm(x,meanlog=pars[1],sdlog=pars[2],log=log)/
      (1-plnorm(L,meanlog=pars[1],sdlog=pars[2]))
  }
  return(ret)
}
```

and then the neg-log-likelihood:

```
nlglik=function(pars,x,L){
  return(
    -sum(dtrlnorm(x=x,pars=pars,L=L,log=TRUE))
    )
}
```

The Rcpp code that calculates the truncated distribution and log-likelihood is below, and is saved as `cppExample3.cpp` file:

```
#include <Rcpp.h>
using namespace Rcpp;


// [[Rcpp::export]]
NumericVector dtrlnormC(NumericVector x, NumericVector pars, double L, int lg) {

  NumericVector out;
  if(lg==1){
    double tmp=log(1-R::plnorm(L,pars[0],pars[1],1,0));
    out= ifelse(x>=L,dlnorm(x,pars[0], pars[1],lg)-tmp,-INFINITY);
  }
  else{
    double tmp=1-R::plnorm(L,pars[0],pars[1],1,0);
    out= ifelse(x>=L,dlnorm(x,pars[0], pars[1],lg)/tmp,0);
  }
  return out;
}

// [[Rcpp::export]]
NumericVector dtrlnormC1(NumericVector x, NumericVector pars, double L, int lg) {
  int n=x.size();
  NumericVector tmp(n);
  NumericVector out(n);

  tmp=dlnorm(x,pars[0], pars[1],lg);

  if(lg==1){
    double cnst=log(1-R::plnorm(L,pars[0],pars[1],1,0));
    for(int i=0;i<n;i++){
      if(x[i]>=L){
        out[i]=tmp[i]-cnst;
      }
      else{
        out[i]=-INFINITY;
      }
    }
  }
  else{
    double cnst=1-R::plnorm(L,pars[0],pars[1],1,0);
    for(int i=0;i<n;i++){
      if(x[i]>=L){
        out[i]=tmp[i]/cnst;
      }
      else{
        out[i]=0;
      }
    }
  }
  return out;
}


// [[Rcpp::export]]
double nlglikC(NumericVector pars, NumericVector x, double L){

  return -sum(dtrlnormC(x,pars, L,1));
}

// [[Rcpp::export]]
double nlglikC1(NumericVector pars, NumericVector x, double L){

  return -sum(dtrlnormC1(x,pars, L,1));
}
```

Note that two implementations of the same `dtrlnorm*` procedure have been reported: the first is slightly less performant, but more "concise", since it uses some facilities of Rcpp *Sugar* that allows the developer to use vector functions (see the `ifelse()` function). The only difference between the two `nlglik*` functions, instead, is on the call to `dtrlnormC` or `dtrlnormC1`.

Now let's compile the above code:

```
sourceCpp('cppExample3.cpp')
```

Now, suppose that a sample from a left truncated lognormal distribution has been produced:

```
set.seed(1000)
dt=rlnorm(n=1000,meanlog=9,sdlog=2)
L=4000
dt=dt[dt>L]
```

We can produce some comparison bechmarks between R and C++ implementations

```
# Start values for optimization process:
init=c(mean(log(dt)),sd(log(dt)))

microbenchmark(
  dtrlnorm(init,x=dt,L=L,log=FALSE),
  dtrlnormC(init,x=dt,L=L,lg=FALSE),
  dtrlnormC1(init,x=dt,L=L,lg=FALSE),
  dtrlnorm(init,x=dt,L=L,log=TRUE),
  dtrlnormC(init,x=dt,L=L,lg=TRUE),
  dtrlnormC1(init,x=dt,L=L,lg=TRUE)
)

## Unit: microseconds
##                                        expr   min     lq median     uq
##   dtrlnorm(init, x = dt, L = L, log = FALSE) 84.52  86.06  88.40  90.73
##   dtrlnormC(init, x = dt, L = L, lg = FALSE) 65.21  66.28  67.72  68.72
##  dtrlnormC1(init, x = dt, L = L, lg = FALSE) 57.78  59.21  60.58  61.82
##    dtrlnorm(init, x = dt, L = L, log = TRUE) 98.00 100.92 102.91 105.77
##    dtrlnormC(init, x = dt, L = L, lg = TRUE) 61.53  63.08  64.52  65.60
##   dtrlnormC1(init, x = dt, L = L, lg = TRUE) 54.25  56.06  57.28  58.87
##     max neval
##  228.91   100
##   97.59   100
##   77.67   100
##  121.96   100
##   93.06   100
##   71.54   100

microbenchmark(
  nlglik(init,x=dt,L=L),
  nlglikC(init,x=dt,L=L),
  nlglikC1(init,x=dt,L=L)
)

## Unit: microseconds
##                           expr    min     lq median     uq    max neval
##    nlglik(init, x = dt, L = L) 100.62 103.30 106.29 111.64 196.46   100
##   nlglikC(init, x = dt, L = L)  61.57  62.81  64.42  66.17  91.51   100
##  nlglikC1(init, x = dt, L = L)  54.58  56.30  57.27  58.77  76.16   100
```

Note that, at the best, the C++ implementation requires about one-half of time required by the "pure R" code.

An estimate of lognormal parameters may be obtained via maximum likelihood. Below an implementation of a function that performs such as calculation using `optim()`:

```
lnMLE=function(x,nllik,L=0,method="Nelder-Mead",
               init=c(mean(log(x)),sd(log(x)))){
  invisible(
    optim(par=init,fn=nllik,method=method,x=x,L=L))
}
```

Note that the function takes as a parameter the specific instance of neg-log-likelihood function to be used, and that the C++ compiled Rcpp functions can be passed to `optim()` like all standard R functions.

And now the estimates:

```
ests=lnMLE(x=dt,nllik=nlglik,L=L)
estsC=lnMLE(x=dt,nllik=nlglikC,L=L)
estsC1=lnMLE(x=dt,nllik=nlglikC1,L=L)

ests
```

```
## $par
## [1] 9.418 1.718
##
## $value
## [1] 7373
##
## $counts
## function gradient
##       51       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
ests$par-estsC$par
```

```
## [1] 0 0
```

```
ests$par-estsC1$par
```

```
## [1] 0 0
```

```
estsC$par-estsC1$par
```

```
## [1] 0 0
```

```
ests$value-estsC$value
```

```
## [1] -5.457e-12
```

```
ests$value-estsC1$value
```

```
## [1] -5.457e-12
```

```
estsC$value-estsC1$value
```

```
## [1] 0
```

Only very small differences appear between the different implementations.

Now a last benchmark on MLE procedures:

```
microbenchmark(
  lnMLE(x=dt,nllik=nlglik,L=L,init=init),
  lnMLE(x=dt,nllik=nlglikC,L=L,init=init),
  lnMLE(x=dt,nllik=nlglikC1,L=L,init=init),times=1000
)
```

```
## Unit: milliseconds
##                                                expr   min    lq median
##    lnMLE(x = dt, nllik = nlglik, L = L, init = init) 5.156 5.284  5.347
##   lnMLE(x = dt, nllik = nlglikC, L = L, init = init) 3.230 3.324  3.354
##  lnMLE(x = dt, nllik = nlglikC1, L = L, init = init) 2.905 2.985  3.016
##     uq   max neval
##  5.402 8.243  1000
##  3.389 6.859  1000
##  3.054 5.200  1000
```

The C++ implementation requires about one half of time required by the R code.

And now, we will use the `truncgof` package to apply a right-tailed AD2 test on simulated data (see the `ad2up.test()` help for info about the function):

```
require(truncgof)
```

```
## Loading required package: truncgof
## Loading required package: MASS
##
## Attaching package: 'truncgof'
##
## The following object is masked from 'package:stats':
##
##     ks.test
```

```
set.seed(100000)
system.time(print(ad2up.test(x=dt,distn="plnorm",fit=ests$par,H=L,
                       sim=1000,tol=0,
                       estfun="as.list(lnMLE(x=dt, nllik=nlglik, L = H)$par)")))
```

```
##
##  Quadratic Class Anderson-Darling Upper Tail Test
##
## data:  dt
## AD2up = 5.373, p-value = 0.58
##
## treshold = 4000, simulations: 1000
##    user  system elapsed
##   5.804   0.008   5.817
```

```
set.seed(100000)
system.time(print(ad2up.test(x=dt,distn="plnorm",fit=estsC$par,H=L,
                       sim=1000,tol=0,
                       estfun="as.list(lnMLE(x=dt, nllik=nlglikC, L = H)$par)")))
```

```
##
##  Quadratic Class Anderson-Darling Upper Tail Test
##
## data:  dt
## AD2up = 5.373, p-value = 0.58
##
## treshold = 4000, simulations: 1000
```

```
##    user  system elapsed
##   3.910   0.006   3.919
```

```
set.seed(100000)
system.time(print(ad2up.test(x=dt,distn="plnorm",fit=estsC1$par,H=L,
                       sim=1000,tol=0,
                       estfun="as.list(lnMLE(x=dt, nllik=nlglikC1, L = H)$par)")))
```

```
##
##  Quadratic Class Anderson-Darling Upper Tail Test
##
## data:  dt
## AD2up = 5.373, p-value = 0.58
##
## treshold = 4000, simulations: 1000
```

```
##    user  system elapsed
##   3.530   0.007   3.541
```

All the approaches produce the same results, but the best Rcpp implemetation requires a few more than one half of the “pure R” computational time.