Chapter 8
Programming in R

Prerequisites and goals of this chapter

- Read all previous chapters first. A neophyte user can skim through this chapter on first reading. Indeed, it is well known that programming in a language requires a more advanced level than using a language.
- The aim of this chapter is to give the user the opportunity to develop new functions; in R, this corresponds to extending the language. The user can thus complete his comprehension of how R works.

SECTION 8.1

Preamble

The strength of the R system is that it includes a real programming language. We shall see that it offers very original programming concepts. The concept of objects is very present in R. Object-oriented programming as used in R is transparent for the user, in the sense that you do not need to understand the theory in order to use it. The same cannot be said for the developer who wishes to respect the spirit of R.

Practical Problem

As an example, this chapter will tackle the resolution of the following practical problem. Suppose that some users, beginners in R, wish to discover programming in R by developing a few functions relative to the well-known least squares methods,[1] in the context of simple linear regression. He soon realizes that two specific tasks

[1] See for example `http://en.wikipedia.org/wiki/Ordinary_least_squares`.

P. Lafaye de Micheaux et al., *The R Software: Fundamentals of Programming and Statistical Analysis*, Statistics and Computing 40, DOI 10.1007/978-1-4614-9020-3_8,
© Springer Science+Business Media New York 2013

are of interest to him: first, output a summary with estimations and the coefficient of linear correlation; second, draw a scatter plot with the regression line. With his experience from previous chapters, this user finds it easy to produce these results from the command line. However, he/she would like to avoid having to type in several lines of commands every time he/she wishes to see the result of these two tasks, and so would like to develop two functions, easier to apply in a daily use of R. To this end, he/she will have the help of a more advanced user who can advise him/her every time he/she encounters a difficulty.

This practical problem should help the reader understand the use of the notions presented in this chapter.

SECTION 8.2

Developing Functions

First of all, let us introduce some basic theoretical elements to explain how to create a function in R.

8.2.1 Quick Start: Declaring, Creating and Calling Functions

Declaring a function is done with the following general form:

```
function(<list of arguments>) <body of the function>
```

where

- `<list of arguments>` is a list of named (formal) arguments.
- `<body of the function>` represents, as the name suggests, the contents of the code to execute when the function is called.

Here is an example of function declaration:

```
> function(name) cat("Hello",name,"!")
function(name) cat("Hello",name,"!")
```

For R, a function is a specific object. Creating a function thus corresponds to affecting the object "R function" to a variable, the name of which corresponds to the function itself. For example, to create the function `hello()`, you can proceed as follows:

```
> hello <- function(name) cat("Hello",name,"!")
> hello
function(name) cat("Hello",name,"!")
```

For this function to be executed, the user needs to call the function, followed by the effective arguments listed in brackets. Recall that an **effective argument** is the value affected to a formal argument. We will use the terms calling argument and input argument as synonyms of effective argument.

```
> hello("Peter")
Hello Peter !
```

8.2.2 Basic Concepts on Functions

8.2.2.1 Body of a Function

The body of a function can be a simple R instruction, or a sequence of R instructions. In the latter case, the instructions must be enclosed between the characters { and } to delimit the beginning and end of the body of the function. Several R instructions can be written on the same line as long as they are separated by the character ;. When the body of the function includes several R instructions written on the same line, do not forget to enclose them between characters { and }. Recall that on a line, any code written after the character # is not interpreted by R and is taken to be a comment.

```
> hello <- function(name) {
+   # Convert the name to upper case.
+   name <- toupper(name)
+   cat("Hello",name,"!")
+ }
> hello("Peter")
Hello PETER !
```

8.2.2.2 List of Formal and Effective Arguments

In this section, we describe how to declare the list of formal arguments when defining a function and how to input the list of effective arguments when calling a function.

Declaring a Function

When declaring a function, **all arguments are identified by a unique name**. Each argument can be associated with a default value. To specify a default value, use the character = followed by the default value, as when declaring a list object (`list()`). When the function is called with no effective argument for that argument, the default value will be used. We have used this functionality many times in previous chapters, but we now know how to include it when developing new functions. Here is an example:

```
> hello <- function(name="Peter") cat("Hello",name,"!")
> hello()
Hello Peter !
```

It seems useful to explain the difference between calling the name of the function hello and calling the function followed by brackets: hello(). The first form will display the contents of the function, as with any other R object, whereas the second form will call the function (in this case, with no argument specified). To execute a function, you always have to add brackets and list the effective arguments if necessary.

Naming Effective Arguments

In R, an effective argument can be entered by adding the name of the formal argument. Of course, this is of little interest when the function only depends on a single formal argument. Let us add to our function hello() the possibility of choosing a language, and see a few calls of this function.

```
> hello <- function(name="Peter",language="eng") {
+   cat(switch(language,fr="Bonjour",sp="Hola",eng="Hello"),
     name,"!")
+ }
> hello()
Hello Peter !
> hello(name="Ben")
Hello Ben !
> hello(language="fr")
Bonjour Peter !
```

This functionality, combined with the ability to specify default values,[2] allows the developer to define a function with an important list of formal arguments corresponding to call options. Users can then call this function without needing to input all effective arguments. For example, they can affect a value to the last formal argument without having to type in all the other effective arguments. This way, a single function can be used for what would have otherwise required several functions. This is a true specificity[3] of R, which allows an innovative programming mode. For example, read the help file on the functionalities of the function seq() with the various arguments by, length.out and along.with.

Partial Naming of Effective Arguments

In the same context, a second functionality of R is that it allows calling a function without typing in the complete name of a formal argument. Consider the following calls of the function hello():

```
> hello(lang="eng")
Hello Peter !
> hello(l="eng")
Hello Peter !
> hello(l="e")
 Peter !
```

[2] The function missing() is also very useful for this kind of programming.

[3] It should be noted that many programming languages do not have this functionality.

The rule for determining the formal argument corresponding to a partial name is: in the ordered list of formal arguments of the function, the selected formal argument is the **first** formal argument for which there is a match between the first letters of the argument name and the partial name given by the user.

List of Supplementary Arguments "..."

You can give a list of supplementary arguments with the syntax .... When calling the function, all "named" arguments which are not in the list of formal arguments are grouped in the structure .... In the body of the function, the user can then use the syntax ... as if copy-pasting the list of supplementary named arguments. This begs for an example:

```
> test.3points <- function(a="foo",...) print(list(a=a,...))
> test.3points("bar",b="foo")
$a
[1] "bar"
$b
[1] "foo"
```

Generally speaking, a rule of thumb for using the list of supplementary arguments ... in the body of a function is that it should be used as an argument of one or several internal function calls.

Advanced users

When ... is included in a list of arguments and is not in last position, "partial naming of arguments" will not work for all arguments after .... Indeed, a partial formal argument name is then considered as a formal argument in the supplementary list.

```
> test.3points <- function(aa="foo",...,bb="bar") {
+                    print(list(aa=aa,...,bb=bb))}
> test.3points(a="bar",b="foo")
$aa
[1] "bar"
$b
[1] "foo"
$bb
[1] "bar"
```

Note that the value of the formal argument aa has been modified, but that bb did not change its value. The formal argument b was created. To change the value of the second formal argument bb, you need to use the complete name.

```
> test.3points(a="bar",bb="foo")
$aa
[1] "bar"
$bb
[1] "foo"
```

A keen user of partial names might be surprised by the following output when using the function `paste(..., sep = " ", collapse = NULL)` if he/she had taken the liberty of using the partial name (`col`) of the formal argument `collapse`:

```
> paste(c("foo","bar"),col=", ")
[1] "foo , " "bar , "
```

Since partial naming is ineffectual, `col` is considered as a second vector to paste, and the default options of the function `paste()` are used (i.e. `sep=" "` and `collapse=NULL`). To get the desired output, you need to use the complete name of the formal argument `collapse`.

```
> paste(c("foo","bar"),collapse=", ")
[1] "foo, bar"
```

Tip

Generally speaking, when you call a function, you need to specify the value of all formal arguments for which no default value is defined. If you do not, an error occurs. There are however two exceptions. The first corresponds to the case where the argument is not used in the body of the function; this is of course useless and is probably due to a programming mistake. The second exception is when the developer allowed for this case in the body of the program, with the function `missing()`.

```
> hello <- function(name) {
+   if(missing("name")) name <- "Peter"
+   cat("Hello",name,"!")
+ }
> hello()
Hello Peter !
```

8.2.2.3 Object Returned by a Function 115

The function `hello()` above does not return any object. It simply produces a 116
display on the screen. 117

```
> res <- hello()
Hello Peter !
> res
NULL
```

In previous chapters, we have often used R functions and saved the result as a 118
variable (e.g., `x <- c(1,5,3)`, where the result of the base function `c()` is affected 119
to the variable `x`). Since we are now interested in developing functions, let us exam- 120
ine how to create a function which returns an object (a result that is not ephemeral). 121

A general rule to return an object is to use the function `return()`. This instruction halts the execution of the code of the body of the function and returns the object between brackets. Here is an example:

```
> hello <- function(name="Peter") {
+   return(paste("Hello",name,"!",collapse=" "))}
> hello()
[1] "Hello Peter !"
> message <- hello()
> message
[1] "Hello Peter !"
```

The first call of the function returns the string of characters object without affecting it to a variable. The result is thus displayed on the screen, as if the user had entered in the command line the object returned by the function. The second call does not produce any display: the result of the function is redirected to the variable `message`, as the last instruction above shows.

Note

It is possible to return an object without using the function `return()`. The rule is then that the returned object is the last object manipulated in the last instruction of the body of the function (i.e. just before exiting the function). In the previous example, we could therefore have omitted the function `return()`

```
> hello <- function(name="Peter") {
+   paste("Hello",name,"!",collapse=" ")}
> hello()
[1] "Hello Peter !"
```

However, we discourage this practice because it does not always work, as shown below where we would expect that the function returns 10:

```
> function.without.return <- function() {
+   for (i in 1:10) x <- i}
> function.without.return()
```

Can you tell whether the following function returns an object? If yes, what is the content of this object?

```
> hello <- function(name="Peter") {
+    msg <- paste("Hello",name,"!",collapse=" ")}
```

What do you think when you see the output below?

```
> hello()
```

There is no display, so it seems that no object is returned. But are you certain when you see the following example?

```
> message <- hello()
> message
[1] "Hello Peter !"
```

The last manipulated object is indeed the variable `msg`. Affecting the output to the variable `message` does store the contents of the variable `msg` from the body of the function. R can sometimes be unsettling, but you will agree that this kind of usage is not rational and a developer would probably never find it useful.

Tip

If you wish to get the same behaviour as in the last example, i.e. that the function does not display anything when called but does return an object, it is more direct to use the function `invisible()`—the name of this function is clear enough.

```
> hello <- function(name="Peter")
+ invisible(paste("Hello",name,"!",collapse=" "))
> hello()
> message <- hello()
> message
[1] "Hello Peter !"
```

8.2.2.4 Variable Scope in the Body of a Function

The notion of variable scope is very important for a language which allows to develop functions. The main point is that variables defined inside the body of a function have a local scope during function execution. This means that a variable inside the body of a function is physically different from another variable with the same name, but defined in the workspace of your R session. Generally speaking, local scope means that a variable only exists inside the body of the function. After the execution of the function, the variable is thus automatically deleted from the memory of the computer. We are now going to modify our function `hello()` by inserting controls of the contents of variables.

```
> message <- "hello Pierre !"
> message # Workspace initialization.
[1] "hello Pierre !"
> hello <- function(name="Peter",message="hello") {
+   print(message)
+   message <- paste(message,name,"!",collapse=" ")
+   print(message)
+   invisible(message)
+ }
> hello()
[1] "hello"
[1] "hello Peter !"
> message # Workspace has not been modified!
[1] "hello Pierre !"
> message <- hello()
[1] "hello"
[1] "hello Peter !"
```

```
> message # Workspace has been modified!
[1] "hello Peter !"
> message <- hello(message="Welcome")
[1] "Welcome"
[1] "Welcome Peter !"
> message # Workspace has been modified again!
[1] "Welcome Peter !"
```

A quick comment on the arguments of the function: contrary to what you might think, the variables `name` and `message` are not directly evaluated (initialized to the calling value or to the default value) before the execution of the body of the function. They are only initialized when they are first used in the body of the function. Recall that the function `missing()` is used to test whether a formal argument has been defined when calling the function. The only way for this functionality to be operational is by not evaluating the list of formal arguments at the beginning of the body of the function. Similarly, at the beginning of the body of the function, it is possible to get the effective call (with the completed list of arguments) by using the function `match.call()`.

```
> test.call <- function(aa="bar",... ,bb="foo") {
+   print(match.call())}
> test.call(a="foo",b="bar")
```

Advanced users

The last function creation may not seem very useful, but once you are an advanced R developer, you might find a use to the result of the function `match.call()`. We shall not give details, but only a taste of what can be done in R. We shall modify the last function so that it returns the arguments split into two lists: one (called `function`) of effective arguments associated with formal arguments and one (called `misc`) of supplementary effective arguments. Note how partial naming of arguments is managed.

```
> test.call <- function(aa="bar",...,bb="foo") {
+   args <- as.list(match.call())[-1]
+   inside <- names(args) %in% names(list(...))
+   list(funct=args[!inside],misc=args[inside])
+ }
> test.call(a="foo",b="bar")
$funct
$funct$aa
[1] "foo"
$misc
$misc$b
[1] "bar"
```

A few lines of code are enough to get the result: introspection is easy in R and has many other features in the same context. We are not trying to get you to delve straight away into this kind of development, but wish to point out the possibilities of the language.

8.2.3 Application to the Practical Problem

After these theoretical explanations, our beginner user tries the following function codes for simple linear regression.

```
mysummary.reg1 <- function(y,x) {
  aEst <- cov(x,y)/var(x)
  bEst <- mean(y)-aEst*mean(x)
  return(list(aEst=aEst, bEst=bEst,cor=cor(x,y)))
}

mydisplay.reg1 <- function(y,x) {
  aEst <- cov(x,y)/var(x)
  bEst <- mean(y)-aEst*mean(x)
  plot(x,y)
  abline(a=bEst,b=aEst)
}
```

Note

Note that in old versions of R, you could write `return(aEst=aEst, bEst=bEst,cor=cor(x,y))` but that this usage will be deprecated in future versions.

After loading these functions with a copy–paste or with the command `source()`, the user tests an uninteresting example.

```
> y <- rnorm(10);x <- 1:10
> mysummary.reg1(y,x)
$aEst
[1] -0.1019453
$bEst
[1] 0.7822879
$cor
[1] -0.4198245
```

The instruction `mydisplay.reg1(y,x)` produces Fig. 8.1 on page 211. We shall see later on how these functions can be enriched.

8.2.4 Operators

Calling a function under the form <`function`>(<`list of call arguments`>) is not always easy. An example is the function `seq()`. Of these two equivalent forms, which one do you prefer?

```
> seq(1,3)
[1] 1 2 3
```

```
> 1:3
[1] 1 2 3
```

You probably prefer the second form, since it is more synthetic (no brackets) and is thus easier to manipulate, for example, when using indices (of vectors, matrices, etc.). This form corresponds to an operator. R uses operators internally.

There are two forms of operators:

- **Unary operator** (one argument) : `<operator> <argument1>`
- **Binary operator** (two arguments) : `<argument1> <operator> <argument2>`

where `<operator>` is the operator, and `<argument1>` and `<argument2>` are the effective arguments of the operator. Here is a partial list of operators used internally by R:

```
+, -, *, /, ^, %%, %/%, &, |, !, ==, !=, <, <=, >=, >.
```

A priori, these operators cannot be modified by the user.[4] It is however possible to define extra operators. They are of the form `%<operator>%` and some are already available in the base system, for example, `%in%` and `%o%` (seen in Chap. 5).

Tip

To display the source of the function (the operator) `%in%`, use the instruction `get("%in%")`. You can see that it uses the function `match()` which you may find useful.

Suppose we wish a more synthetic way to concatenate strings of characters, which is normally done with the function `paste()`.

```
> "%+%" <- function(ch1,ch2) paste(ch1,ch2,sep="")
> name <- "Peter"
> "The life of " %+% name %+% " is beautiful!"
[1] "The life of Peter is beautiful!"
> # This is a simplification of:
> paste("The life of ", name ," is beautiful!",sep="")
[1] "The life of Peter is beautiful!"
```

Note that since the name of the function is not alphanumeric, it has to be put between quotation marks. It is of course up to you whether you prefer one or the other form. We are not trying to diminish the usefulness of the function `paste()`, which is a much richer function than the simple operator `%+%` we have created (the creation actually used the function `paste()`). We are rather trying to show the flexibility of R which allows, with a simple function definition, a simplification of the calling syntax.

[4] In fact, this group of operators can be used by a user when developing a new class of objects. But this matter is too advanced for this book!

Tip

You can use operators to define operations on sets, such as those presented on p. 99. For example, the union between two sets A and B can be defined as

```
> "%union%" <- function(A,B) union(A,B)
> A %union% B
[1] 4 6 2 7 1 3
```

8.2.5 R *Seen as a Functional Language*

R is a functional language in the sense that almost any code execution in R is done by calling functions, possibly scattered with control structures. In fact, you may be surprised to learn that the following features of R are also controlled by functions. We have seen that simply calling an R object results in the display of its contents. In fact, in such an instruction, R calls (without notifying the user) the function `print()` with effective argument the name of the object. Because this function is often used in R, it has a particular status; we shall discuss this further later on. All affectation operations (i.e. instructions with <-) are handled by functions whose names include (no surprise here) the distinctive sign <-[5]. Developing and maintaining the R system can be summarized as the construction of a range of functions. First are the basic functions, included in the basic installation of R. Usually, they cannot be modified by the user[6], and even when they can be, we strongly advise against it; let your system become unusable. Second are the functions developed directly in R[7] by any user. Many functions are made available by the community of R developers through a system of packages (more on this later).

SECTION 8.3

† Object-Oriented Programming

In this section, we shall view an object as more than a quantity that can be saved and reused. We shall come closer to the spirit of the R language by looking at the internal object-oriented mechanism which governs most of its use. The incredible part is that the user does not need to worry about knowing the internal workings of R. According to us, this is a strong point of R. Nonetheless, this section should

[5] To see this, type in the command line `apropos("<-")`.

[6] The core of R is developed in the C language for obvious reasons of speed of execution, which makes it rather reactive when used in the command line.

[7] To speed up execution, it is usually possible to convert an R function into C and then to call it from R via the C API.

help users better understand how R proposes results. We expect this will lead to a less "random" and more controlled use of R.

8.3.1 *How the Internal Object-Oriented Mechanism Works*

8.3.1.1 Class of an Object and Declaring an Object

What matters in R is specifying the class of an object with the function `"class<-"()`. Recall that the function `class()` is used to check the class of an object.

```
> obj <- 1:10
> class(obj)
[1] "integer"
> class(obj) <- "MyClass"
> class(obj)
[1] "MyClass"
> class(obj) <- "OtherClass"
> obj
 [1]  1  2  3  4  5  6  7  8  9 10
attr(,"class")
[1] "OtherClass"
```

The object `obj` of class `integer` is now an object of class `OtherClass`. The last display of the object `obj` indicates the class of the object, where `attr` stands for attribute. We shall come back to the notion of attributes at the end of this chapter. For now, it is enough to understand the meaning of the display `attr(,"class")` which is literally the "class attribute".

Advanced users

That said, the above is not quite true: the object `obj` has kept the characteristic of also being of the `integer` class, as the following output shows:

```
> obj*2
 [1]  2  4  6  8 10 12 14 16 18 20
attr(,"class")
[1] "OtherClass"
```

Indeed, all the elements of the vector `obj` have been multiplied by 2. We hope that in future versions of R, the output of the function `class()` applied to such an object will be similar to `[1] "OtherClass" "integer"`, which would better show the true nature of the object.

There are two ways of knowing whether an object is of a given class:

```
> class(obj)=="MyClass"
[1] FALSE
```

```
> inherits(obj,"MyClass")
[1] FALSE
```

The function `inherits()` should be preferred, as we shall see when we consider polymorphic objects with several classes.

Tip

To see the class of the function `function()`, you can use this instruction:

```
> class(function() {})
[1] "function"
```

For the function `":"()` , use `class(get(":"))`.

8.3.1.2 Declaring Objects and Using Methods

The mechanism for object-oriented programming is rather simple and original in R, compared to many other languages. To illustrate this mechanism, examine the most used example in R: the display of an object with the function `print()`. Examine the following R outputs:

```
> vect <- 1:10
> class(vect)
[1] "integer"
> vect
 [1]  1  2  3  4  5  6  7  8  9 10
> print(vect)
 [1]  1  2  3  4  5  6  7  8  9 10
```

No surprises so far, although it is worth pointing out that simply entering an R object in the command line seems to provoke a call to the function `print()` with the given object as effective argument. The next example confirms this idea[8]: it displays an object of the class `formula`, characterized by the tilde symbol ($\sim$). In this example, we save in the variable `form` the formula expressing the relationship between `y` and `x`. Note that the objects `y` and `x` do not need to exist, since no evaluation is done when a formula is defined.[9]

```
> form <- y~x
> class(form)
[1] "formula"
> form
y ~ x
> print(form)
y ~ x
```

[8] In fact, for auto-printing base objects (vectors, matrices, lists, etc.) in the console, R does not use the `print()` function, but calls a C function named `PrintValueEnv`, which is not directly available to the user.

[9] No further details are needed for now; we shall come back to this very original class of objects.

Note that the function `print()` works differently for different classes of objects. For the variable `form` (of class `"formula"`), `print()` returned `y~x`, which is the instruction to the right of the affectation arrow. For the variable `vect`, calling `print()` returns `[1] 1 2 3 4 5 6 7 8 9 10` when we might have expected it to display `1:10`. Here is the code of the function `print()`:

```
> print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
```

The body of this function indicates that the function `UseMethod()` must be executed. This function is a *generic function* in R. Like an airport traffic control tower, it is used to redirect the object, according to its class, to the correct function call. In the last example, this corresponds to calling the display function associated with the class `formula` of the form `print.formula()`. In the object-oriented programming vernacular, such functions, of the general type <`method`>.<`class`>, are called *methods*. This explains the name of the function `UseMethod()` in the body of the generic function `print()`.

Here is what happens in the backstage to simply display the object `form` :

```
> form # Calls the function print(),
       # which calls the function print.formula().
y~x
> print.formula(form)
y~x
```

Advanced users

To check how easy it is to change the general behaviour of R by changing one function, we are going to redefine the display function for the class `formula`. We are simply going to keep the standard display and add the string of characters `"formula:"`.

```
> print.formula <- function(obj,...) {
+ cat(paste("formula:",paste(sapply(obj[c(2,1,3)],
+                  as.character),collapse="")))
+ invisible(obj)
+ }
> y~x
formula: y~x
```

If you are a beginner in R, you should not try to understand the details of the R code leading to this result. Although the code seems simple, understanding it requires notions which we cannot go into in this book. Once again, the aim is rather to reveal the introspective power of R, since even its base elements can be manipulated.

To restore the initial behaviour of R for displaying formulae, you will have guessed that it suffices to delete the new function `print.formula()` with the command line instruction `rm(print.formula)`. We shall not delete it yet, because we need this behaviour later on.

If you have understood the way the function `print()` works, you might expect that there exists a function `print.integer()`. We can check this:

```
> print(vect)
 [1]  1  2  3  4  5  6  7  8  9 10
> print.integer(vect)
Error in eval(substitute(expr), envir, enclos) :
  could not find function "print.integer"
```

The function `print.integer()` does not exist. In fact, when there is no method associated with a class, R executes the default method, which is of general form `<method>.default`; in this case, `print.default()`. Here is the output of this function for our two examples:

```
> print.default(vect)
 [1]  1  2  3  4  5  6  7  8  9 10
> print.default(form)
y ~ x
attr(,"class")
[1] "formula"
attr(,".Environment")
<environment: R_GlobalEnv>
> # Compare with:
> form
formula: y~x
```

We now have a complete explanation of what happens behind the scenes. We also see that the display of a formula does not use the default method, as the last output suggests.

Tip

Also note that the function `print.default()` is used to display all base objects (or structures) of R when these objects are taken as effective arguments of the `function print()`.

In summary, to define a new family of methods, denoted here by `<method>` (name of the family of methods you wish to create), which can be applied to any type of object, you need to:

- First declare the ***generic function*** in the following form:
  `<method> <- function(obj,...) UseMethod("<method>")`
- Then create a ***method*** `<method>` for a class `<class>`:
  `<method>.<class> <- function(obj,<list of arguments>) <body of the method>`
  where `<list of arguments>` and `<body of the method>` are, respectively, an optional list of formal arguments and the contents of this method, which is nothing else than a function when called in its long version.

Note

Note that when declaring a family of methods, you can dissociate the name of the generic function and the argument of the function `UseMethod()` corresponding to the name of the method to call. Thus, it is easy to define an alias, called `<alias>`, of the last family of methods by simply defining a new generic function:

```
<alias> <- function(obj,...) UseMethod("<method>")
```

As a result, the two command line calls `<method>(<object>)` and `<alias>(<object>)` for an object `<object>` of class `<class>` are equivalent to `<method>.<class>(<object>)`. A rather surprising application is that a method can be translated like this. In the next example, the French *voir* is used as an alias of `print`:

```
> voir <- function(obj,...) UseMethod("print")
> voir(vect)
 [1]  1  2  3  4  5  6  7  8  9 10
> voir(form)
formula: y~x
> rm(print.formula) # Remove our method to return
                    # to the normal mode.
> voir(form)
y ~ x
> form
y ~ x
```

8.3.2 Back to the Practical Problem

The user realizes that he/she has repeated the execution of the estimations of a and b twice when creating the functions `mydisplay.reg1()` and `mysummary.reg1()` introduced in Sect. 8.2.3 (lines 2 and 3, and lines 8 and 9). He asks advice from a more advanced user, who suggests using the concept of object-oriented programming. He/she proposes to create a function[10] to return an object of class `reg1`, so that it can be reused thereafter as first calling argument for any method of the said class.

```
reglin <- function(y,x) {
  aEst <- cov(x,y)/var(x)
  bEst <- mean(y)-aEst*mean(x)
  reg <- list(y=y,x=x,aEst=aEst,bEst=bEst)
  class(reg) <- "reg1"
  return(reg)
}
```

[10] This kind of function is often called a constructor in object-oriented programming.

They now define the method mydisplay.reg1() which can be used on any object of class reg1.

```
mydisplay.reg1 <- function(reg) {
  plot(reg$y,reg$x)
  abline(a=reg$bEst,b=reg$aEst)
}

mysummary.reg1 <- function(reg) return(reg)
```

They try a few tests:

```
> reg <- reglin(y,x)
> mysummary(reg)
Error in eval(substitute(expr), envir, enclos) :
  could not find function "mysummary"
> mydisplay(reg)
Error in eval(substitute(expr), envir, enclos) :
  could not find function "mydisplay"
```

The user did not expect such errors, so he/she checks that the function is well defined:

```
> mysummary.reg1(reg)
$y
 [1]  1.8920106  0.3978771 -0.3970281 -0.2799578  0.7851185
 [6] -0.2103208  0.1921150 -0.2647256 -0.5013911  0.6021898
$x
 [1]  1  2  3  4  5  6  7  8  9 10
$aEst
[1] -0.1019453
$bEst
[1] 0.7822879
attr(,"class")
[1] "reg1"
```

The advanced user points out the mistake: the generic functions mysummary and mydisplay have not been declared and are not standard, unlike a few others such as print() and summary().

```
mysummary <- function(x,...) UseMethod("mysummary")
mydisplay <- function(x,...) UseMethod("mydisplay")
```

The previous instructions now work:

```
> mysummary(reg)
$y
 [1]  1.8920106  0.3978771 -0.3970281 -0.2799578  0.7851185
 [6] -0.2103208  0.1921150 -0.2647256 -0.5013911  0.6021898
$x
 [1]  1  2  3  4  5  6  7  8  9 10
$aEst
[1] -0.1019453
```

```
$bEst
[1] 0.7822879
attr(,"class")
[1] "reg1"
> mydisplay(reg)
```

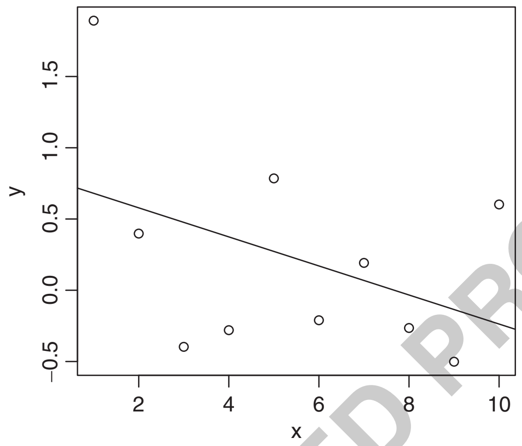


Fig. 8.1: Result of the call of the function `mydisplay.reg1()`

Since the method `print.reg1()` has not been defined, you may wonder what would happen when we simply enter the name of the object. 327 328

```
> reg
$y
 [1]  1.8920106  0.3978771 -0.3970281 -0.2799578  0.7851185
 [6] -0.2103208  0.1921150 -0.2647256 -0.5013911  0.6021898
$x
 [1]  1  2  3  4  5  6  7  8  9 10
$aEst
[1] -0.1019453
$bEst
[1] 0.7822879
attr(,"class")
[1] "reg1"
```

We already knew that the method `print.default()` is called in such cases. 329

8.3.3 Information About Methods 330

To get information about methods, R has the function `methods()`: 331

```
> methods("formula") # Or more directly methods(formula).
[1] formula.character*  formula.data.frame* formula.default*
[4] formula.formula*    formula.glm*        formula.lm*
[7] formula.nls*        formula.terms*
```

```
   Non-visible functions are asterisked
> methods(class="formula")
 [1] [.formula*             aggregate.formula*
 [3] alias.formula*         all.equal.formula
 [5] ansari.test.formula*   bartlett.test.formula*
 [7] boxplot.formula*       cdplot.formula*
 [9] cor.test.formula*      deriv.formula
[11] deriv3.formula         fligner.test.formula*
[13] formula.formula*       friedman.test.formula*
[15] ftable.formula*        getInitial.formula*
[17] kruskal.test.formula*  lines.formula*
[19] mood.test.formula*     mosaicplot.formula*
[21] pairs.formula*         plot.formula*
[23] points.formula*        ppr.formula*
[25] prcomp.formula*        princomp.formula*
[27] print.formula          quade.test.formula*
[29] selfStart.formula*     spineplot.formula*
[31] stripchart.formula*    t.test.formula*
[33] terms.formula          update.formula
[35] var.test.formula*      wilcox.test.formula*
   Non-visible functions are asterisked
```

Warning

Do not confuse the two uses. The first instruction outputs all methods (of the form <`method`>.<`class`>) associated with the generic function `formula`. The second instruction gives all methods for the class `formula`.

Here are a few examples to better understand the distinction between the two 332
uses of the function `methods()`. 333

```
> class(y~x)
[1] "formula"
> update(y~x,.~.+z) # Apply the method update() to an
                    # object of class formula.
y ~ x + z
> update.formula
function (old, new,...)
{
    tmp <-.Internal(update.formula(as.formula(old),
                                  as.formula(new)))
    out <- formula(terms.formula(tmp, simplify = TRUE))
    return(out)
}
<environment: namespace:stats>
> form <- "y~x"
> class(form)
[1] "character"
> formula(form)
y ~ x
> formula.character
Error: object "formula.character" not found
```

Tip

Functions followed with an asterisk can be executed, but the body of the function cannot be visualized. You can however use the function `getAnywhere()`.

```
> getAnywhere(formula.character)
A single object matching 'formula.character' was found
It was found in the following places
  registered S3 method for formula from namespace stats
  namespace:stats
with value
function (x, env = parent.frame(), ...)
{
    ff <- formula(eval(parse(text = x)[[1L]]))
    environment(ff) <- env
    ff
}
<environment: namespace:stats>
```

8.3.4 *Inheriting Classes*

In the context of our practical problem, the advanced user informs the beginner user that R already has a set of functions to manage linear models. Indeed, the function `lm()` is dedicated to this kind of treatment (as we shall see in Chap. 14). However, he/she adds that to his knowledge, no functions exist to perform the specific treatment they propose. The two users work together to develop an extension; they want to avoid "reinventing the wheel" and make the most of existing functions in R.

In object-oriented programming, the notion of class inheritance seems appropriate for this kind of extension. Inheritance expresses the fact that an object of a certain class can also behave like all objects of supplementary classes. Such a mechanism is available in R, by associating a sequence of classes with an object. Thus, when a method is applied to an object which has a hierarchy of classes, the first class is solicited first. If the method exists for this class, it is executed. Otherwise, R tests whether there is an executable method in the class hierarchy. If there is, that method is executed; otherwise, the default method is executed, as long as it is defined. Finally, if none of the above apply, an execution error is generated. Let us illustrate this notion with the problem of our two users. First, we need to declare the constructor function of the new class `lm1`, which inherits directly from the existing class `lm`.

```
1 lm1 <- function(...) {
2   obj <- lm(...)
3   if(ncol(model.frame(obj))>2) stop("more than one
4           independent variable")
5   class(obj) <- c("lm1",class(obj)) # Or c("lm1","lm")
6   obj
7 }
```

Apply this to the same variables as before.

```
> reg <- lm1(y~x)
> reg
Call:
lm(formula = ..1)
Coefficients:
(Intercept)            x
     0.7823      -0.1019
```

We can see inheritance in action. No method `print.lm1()` is defined, and yet the object is not displayed as with `print.default()`. This is because R already knows the method `print.lm()` and the object `reg` inherits methods from the class `lm`. There are several ways of checking that this object is indeed inheriting from this class; the simplest is visualizing the contents of the `class` attribute with the function `class()`. A developer might prefer the more direct function `inherits()`.

```
> class(reg)
[1] "lm1" "lm"
> inherits(reg,"lm")
[1] TRUE
> print.lm(reg)
Call:
lm(formula = ..1)
Coefficients:
(Intercept)            x
     0.7823      -0.1019
```

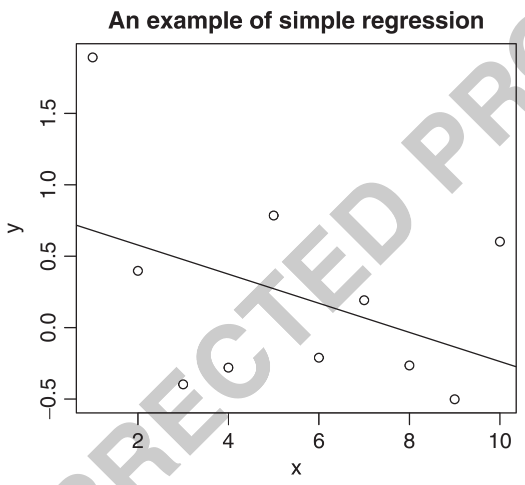
Line 3 (which we shall not comment) in function `lm1()` tests whether the formula is a simple regression model formula. See what happens in this next example:

```
> lm1(y~x+log(x))
Error in lm1(y ~ x + log(x)) : more than one
          independent variable
```

We continue developing functions in the same spirit as

```
plot.lm1 <- function(obj,...) {
  plot(formula(obj),...)
  abline(obj)
}
```

```
> summary(reg)
Call:
lm(formula = ..1)
Residuals:
    Min      1Q  Median      3Q     Max
-0.8735 -0.3772 -0.2060  0.4153  1.2117
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.78229    0.48348   1.618    0.144
x           -0.10195    0.07792  -1.308    0.227
Residual standard error: 0.7077 on 8 degrees of freedom
Multiple R-squared: 0.1763,     Adjusted R-squared: 0.07328
F-statistic: 1.712 on 1 and 8 DF,  p-value: 0.2271
> plot(reg,main="An example of simple regression")
```



377

In the call of summary() above, the method summary.lm1() has not been developed; hence, the standard method summary.lm() is executed. Indeed, the object reg of class lm1 then inherits from the class lm for all standard methods proposed by R to manage linear models. For the call of the method plot(), the freshly created method plot.lm1 is invoked. 378 379 380 381 382

Note

Note that R has a standard method plot.lm() which creates a set of plots for a more detailed analysis of the results (see Chap. 14). We have intentionally changed the default behaviour of R for simple linear regression, but can still access this method by calling it explicitly (plot.lm(reg)).

Advanced users

Object-oriented programming is extremely simple in its conception. There are many object-oriented programming languages. An important difference is that the vast majority offer an encapsulation of object fields and methods; one of the points of this encapsulation is that the fields of an object can be modified within a method. This is not directly possible in R because of the strict local scope of variables inside the code of an R function. The users can however adopt this kind of programming if they want to. Any method `<method>.<class>()` which needs to modify the fields of an object `<object>` (of class `<class>`) must then return the object itself. The user of the generic function `<method>()` can then affect the result to the initial object, as follows:
`<object> <- <method>(<object>)`. However, this risks to slow down execution, all the more if the contents of the object fields are large. This is because the object is completely duplicated. We hope that R developers will one day offer a more elegant standard functionality (analogous to what the majority of object-oriented programming languages offer), whereby only the relevant fields (of which there are usually few) are modified inside the body of the method. When you become an advanced user (as we hope), you will notice that the notion of pointers (which is very common in programming) is not directly offered to R developers (see however the function `tracemem()` as well as Sect. 9.8.2.2, p. 296).

SECTION 8.4

† Going Further in R Programming 383

Before you start programming in a language, it is good to know the spirit in which it was conceived. In this section, we shall explore structures of the R language which you do not need to know when you start using R, but which you will find very useful when you decide to go deeper in your use of R. These elements make R an original and powerful tool. We advise beginner users to skim through this section without trying to master the concepts. All the information in this section is second level, in the sense that a very powerful use of R is possible without it.

8.4.1 R Attributes 391

An R object includes *primary information*, conveyed by the basic structures presented in this book. There is another level of information, which we call *secondary information*. It is attached to an object with attributes and can be accessed with the function `attributes()`.

```
> mat <- matrix(1:10,nrow=2)
> mat
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> class(mat)
[1] "matrix"
> attributes(mat)
$dim
[1] 2 5
```

We shall comment on this output later. For now, let us insist again on the fact that this mechanism is supposed to be transparent for the user, who usually cares more about the contents of the R object. For day-to-day use, we advise you not to change attributes directly. This stand is justified by the existence of many functions to manipulate attributes indirectly. However, a developer who wishes to learn more about the internal workings of R will discover a few supplementary characteristics which usually enlighten the behaviour of the object. We have already indirectly manipulated the attribute `class` with the functions `class()` and `"class<-"()`. We shall also manipulate the three other main attributes: `dim`, `names` and `dimnames`. These are used a lot in the internal management of R. The next example is only interesting to present how to handle attributes. The complementary function `attr()` is used to manipulate a single attribute at a time, whereas the function `attributes()` returns all attributes as an R list.

```
> vect <- 1:10
> attr(vect,"test") # Returns NULL, because vect has no
                    # attribute test.
NULL
> attributes(vect)  # NULL because vect has no attributes.
NULL
> # Affecting an attribute "attrib1" containing the character
  # string "TEST1".
> attr(vect,"attrib1") <- "TEST1"
> attr(vect,"attrib1")
[1] "TEST1"
> # Affecting an attribute "attrib2" containing the vector c(1,3)
> attributes(vect)$attrib2 <- c(1,3)
> attributes(vect)
$attrib1
[1] "TEST1"
$attrib2
[1] 1 3
> attr(vect,"attrib2")
[1] 1 3
> # Modifying attribute "attrib1" and deleting attribute
  # "attrib2"
> attributes(vect)$attrib1 <- 3:1
> attr(vect,"attrib2") <- NULL
> attributes(vect)
$attrib1
[1] 3 2 1
```

```
> # Deleting all attributes at once
> attributes(vect) <- NULL
> attributes(vect)
NULL
```

The attribute access mechanism is simple to use. This example has shown how to change attributes using the functions `"attr<-"()` and `"attributes<-"()`. The value of an attribute can be any R object. Affecting `NULL` to an attribute deletes it. 409 410 411

8.4.1.1 Attribute `class` 412

In Sect. 8.3, we have manipulated the attribute `class` using the functions `class()` and `"class<-"()`. This shows that you do not need to know how to manipulate attributes directly. We return to the example we used, to show that manipulating this attribute is equivalent to using the utility functions `class()` and `"class<-"()`. 413 414 415 416

```
> form <- y~x
> attributes(form)
$class
[1] "formula"
$.Environment
<environment: R_GlobalEnv>
> class(form)
[1] "formula"
> obj <- 1:10
> attr(obj,"class") # No class attribute.
NULL
> class(obj)         # And yet!
[1] "integer"
> attr(obj,"class") <- "MyClass" # Equivalent to class(obj) <-
                                 # "MyClass".
> class(obj)
[1] "MyClass"
```

There is nothing left to say about this attribute, even though it plays a central role in object-oriented programming in R. 417 418

8.4.1.2 Attribute `dim` 419

The attribute `dim` plays an important role in the behaviour of `matrix` and `array` objects. Here is an example with a matrix: 420 421

```
> mat <- matrix(1:12,nrow=2)
> mat
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
> attr(mat,"dim")
[1] 2 6
```

```
> attributes(mat)
$dim
[1] 2 6
> attr(mat,"dim") <- c(3,4)   # Changing shape: 3 rows and 4
                              # columns.
> mat
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> attributes(mat)$dim <- c(2,6) # Back to the initial shape.
> mat
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
```

In this example, changing the attribute dim allowed us to change the shape of 422
the matrix. We have already mentioned that attribute management is meant to be 423
transparent for the user, so you might expect there exist similar functions with more 424
user-friendly names. For this example, we could have used the functions dim() and 425
"dim<-"(): 426

```
> dim(mat)
[1] 2 6
> dim(mat) <- c(1,12) # Changing shape: 1 row and 12 columns.
> mat
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
[1,]    1    2    3    4    5    6    7    8    9    10    11
     [,12]
[1,]    12
> dim(mat) <- c(2,6)  # Back to the initial shape.
```

To really understand how R represents objects such as matrices and arrays, let us 427
analyse the following output: 428

```
> mat
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
> class(mat)
[1] "matrix"
> dim(mat) <- NULL # Or attributes(mat)$dim<-NULL or
                   # attributes(mat) <- NULL.
> mat
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> is.vector(mat)
[1] TRUE
> class(mat)
[1] "integer"
> dim(mat) <- c(2,2,3)
> mat
, , 1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
, , 2
     [,1] [,2]
[1,]    5    7
[2,]    6    8
, , 3
     [,1] [,2]
[1,]    9   11
[2,]   10   12
> is.vector(mat)
[1] FALSE
> class(mat)
[1] "array"
```

When we delete the attribute dim, the object mat becomes a simple vector. When 429
we affect a vector of three integers to this attribute, the object mat becomes an array 430
of dimension 3. The different behaviours of vectors, matrices and arrays thus stem 431
from the value of the attribute dim. 432

Warning

Although the display is the same, a vector and a single-index array are treated differently by R, as shown by these few lines of code:

```
> dim(mat) <- 12
> mat
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> is.vector(mat)
[1] FALSE
> class(mat)
[1] "array"
> identical(mat,1:12)
[1] FALSE
> dim(mat) <- NULL
> mat
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> is.vector(mat)
[1] TRUE
> class(mat)
[1] "integer"
> identical(mat,1:12)
[1] TRUE
```

It looks like we have said everything about the attribute dim, but there is one 433
last application worth noting. The only difference between a vector and a list is that 434
the elements of a vector must all have the same type. Matrices and arrays usually 435
contain elements of the same nature as well; this constraint is very important for 436
matrix operations. But as storage structures, you could imagine extending the matrix 437
and array concepts to lists, by affecting the dim attribute, as is done with vectors. 438
The documentation files for the matrix() and array() instructions show that this 439
is the case, since the first calling argument of these functions can be a list instead of 440

a vector. The next example applies this to a matrix; the same could be done with an array, as long as the number of elements in the list agrees with the dimension. 441 442

```
> lmat <- matrix(list(7,1:2,1:3,1:4,1:5,1:6),nrow=2)
> lmat      # Returns the structure and not the contents, which
            # are too difficult to display.
     [,1]      [,2]      [,3]
[1,] 7         Integer,3 Integer,5
[2,] Integer,2 Integer,4 Integer,6
> dim(lmat)
[1] 2 3
> is.list(lmat)
[1] TRUE
> lmat[1,2] # Extract the element at row 1 and column 2.
[[1]]
[1] 1 2 3
> lmat[,-2] # Extract the submatrix with the second column
            # removed.
     [,1]      [,2]
[1,] 7         Integer,5
[2,] Integer,2 Integer,6
> dim(lmat) <- NULL
> lmat      # This is just a list now.
[[1]]
[1] 7
[[2]]
[1] 1 2
[[3]]
[1] 1 2 3
[[4]]
[1] 1 2 3 4
[[5]]
[1] 1 2 3 4 5
[[6]]
[1] 1 2 3 4 5 6
> is.list(lmat)
[1] TRUE
```

8.4.1.3 Attributes names and dimnames 443

The attribute names plays an important role in naming elements of a list. 444

```
> li <- list(1:3,letters[1:3])
> li
[[1]]
[1] 1 2 3
[[2]]
[1] "a" "b" "c"
> attributes(li)
NULL
> attributes(li)$names <- c("numbers","letters")
> li
$numbers
```

```
[1] 1 2 3
$letters
[1] "a" "b" "c"
```

The first and fourth instructions are thus equivalent to the following, more common declaration:

```
> li <- list(numbers=1:3,letters=letters[1:3]))
```

It is a less useful and lesser known fact that this attribute can also be used on any type of vector.

```
> vect <- 1:3
> attr(vect,"names") <- letters[1:3]
> vect
a b c
1 2 3
> # Or directly
> vect2 <- c(a=1,b=2,c=3)
> vect2
a b c
1 2 3
```

You do not need to manipulate the attribute names directly. Accessing and changing its value can be done explicitly:

```
> names(li)
[1] "numbers" "letters"
> names(li) <- c("num","lett")
> li
$num
[1] 1 2 3
$lett
[1] "a" "b" "c"
> names(vect)
[1] "a" "b" "c"
> names(vect) <- toupper(names(vect))
> vect
A B C
1 2 3
```

For objects with several indices, such as matrices and arrays, index name management is done internally by modifying the attribute dimnames, as shown in this quick example.

```
> mat <- matrix(1:6,nr=2)
> mat
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> attributes(mat) # Can be modified as an attribute.
$dim
[1] 2 3
> rownames(mat)    # Row names.
NULL
> colnames(mat)    # Column names.
```

```
NULL
> dimnames(mat)    # Row and column names as a list.
NULL
> colnames(mat) <- paste("V",1:3,sep="")
> rownames(mat) <- c("a","b")
> mat
  V1 V2 V3
a  1  3  5
b  2  4  6
```

For an array with more than two dimensions, the functions `rownames` and `colnames` are meaningless. You can either modify the attribute `dimnames` directly or use the function `"dimnames<-"()`.

Note

Data frames have a special status. They are defined as lists and are usually manipulated as matrices. The attributes for row and column names are `row.names` and `names` (instead of `col.names`) :

```
> df <- data.frame(a=1,b=1:2)
> df
  a b
1 1 1
2 1 2
> attributes(df)
$names
[1] "a" "b"
$row.names
[1] 1 2
$class
[1] "data.frame"
> names(df)      # As a list.
[1] "a" "b"
> dimnames(df)   # As an array: list of two vectors.
[[1]]
[1] "1" "2"
[[2]]
[1] "a" "b"
> rownames(df)   # As a matrix: accessing the row names.
[1] "1" "2"
> colnames(df)   # As a matrix: accessing the column names.
[1] "a" "b"
```

The last four lines give calls to access these attributes without manipulating them directly. Corresponding forms exist to change their values. Note that the attribute `class()` gives the class of the object.

8.4.2 Other R Objects

It could be said that one of the specificities of R is that the vast majority of quantities manipulated by R are allocated to variables and can thus be reused later on. There are a few exceptions, mostly control structures. R objects are of different types, called classes. We have already encountered object classes used to store common data. There are three other object types we chose to explore as well. Surprisingly, formulae and environments are also objects in R; we shall also introduce R expressions, which are objects in which R code can be stored to be executed at a later time.

8.4.2.1 R Expressions

So far, we have said nothing on structures used to described the syntactic bases of R. Following its philosophy of managing as many components as possible, R can manipulate an R expression and split it into a sequence of atomic entities (such as `call`, `name`…). We only mention these capacities, without going into the details. We shall focus on R expressions which are truly of interest to an R developer. It is difficult to give a rigorous definition of R expressions. We propose the following definition, inspired by command line use of R. An R expression can be seen as R code entered in sequence as command lines until it is executed by the R interpreter (i.e. until the character > is displayed, inviting a new command). This expression can spread over several lines. The function `expression()` is used to declare an R expression when it is used with a single calling argument. It is however possible to give a sequence of expressions, each expression corresponding to one effective argument in the call of a function. An expression object is not evaluated by the R interpreter but can be saved to be evaluated later, as many times as needed. Evaluating an R expression is done with the function `eval()`. All of this is illustrated in this example:

```
> expression(v<-"value")              # The expression v<-"value"
                                      # is not evaluated.
expression(v <- "value")
> v
Error in eval(substitute(expr), envir, enclos) :
object 'v' not found
> expression(v<-"value") -> expr      # Saved in the object expr.
> expr
expression(v <- "value")
> eval(expr)                          # Evaluating expr.
> v                                   # Here is the expected
                                      # result.
[1] "value"
> expression(v<-"value2",v) -> expr   # Equivalent to 2 lines of
                                      # unevaluated commands.
> expr
expression(v <- "value2", v)
> eval(expr)                          # The second instruction
                                      # displays the contents of
                                      # v.
[1] "value2"
```

A developer will find it useful to convert a character string describing R code into an R expression to be evaluated at another time. The function `parse()` is used to this effect:

```
> parse(text='v<-"value"') -> expr
> expr
expression(v<-"value")
attr(,"srcfile")
<text>
> eval(expr)
> v
[1] "value"
```

The formal argument `text` is used here to read a character string, but the first use of the function is to read a file containing R code; the name of the file is given as the first effective argument.

Tip

Here is an example using the functions `eval()` and `parse()`:

```
> for (i in 1:3) eval(parse(text=paste("a",i," <- i",sep="")))
> a2
[1] 2
```

We are now going to manipulate the function `expression()` to describe some of the internal behaviour of R. This will help understand why R is said to be a functional language (i.e. which makes an intensive use of functions). It is surprising how true this is. This first point shows that upon execution, affectation is considered as an operator (a function with two arguments). The first argument corresponds to the variable, the second to the contents.

```
> foo <- "foo"
> foo
[1] "foo"
> "<-"(foo,"foo2")               # Equivalent to: foo <- "foo2"
> foo
[1] "foo2"
> expression("<-"(foo,"foo2"))   # as shown by the output of this
                                 # expression.
expression(foo <- "foo2")
```

We continue our exploration with brackets. One of the uses of brackets is to order execution priorities in an R expression. Again, R treats them as a function.

```
> 30*(10+20)
[1] 900
> 30*"("(10+20)   # This is what is executed behind the scenes.
[1] 900
> expression(30*10+20))
expression(30 * (10 + 20))
> expression(30*"("(10+20))
expression(30 * (10 + 20))
```

The same is true for the notion of expression blocks. An expression block is defined as a sequence of R expressions, grouped between curly bracket delimiters "{" and "}".

```
> {
+ print("line1")
+ print("line2")
+ }
[1] "line1"
[1] "line2"
> "{"(print("line1"),print("line2"))
[1] "line1"
[1] "line2"
> expression({
+   print (       "line1"     ) # This comment is not interpreted.
+
+   # Neither is this comment.
+   print("line2")
+ })
expression({
    print("line1")
    print("line2")
})
> expression( "{"(print("line1"),print("line2")) )
expression({
    print("line1")
    print("line2")
})
```

Note that comments and spaces are ignored by the R interpreter. Note also that to make your code easier to read, you can add as many carriage returns as you wish in a block without any effect on its execution.

8.4.2.2 R Formulae

The formula object is one of the specificities of R. It is mainly used to establish a relationship between two parts, separated with a tilde ∼. Both parts must be R expressions. Keeping in mind what we have learnt about the function `expression()`, we can see how R converts a formula into a "∼"() function upon execution.

```
> y~x
y ~ x
> "~"(y,x)              # Equivalent expression,
y ~ x
> expression("~"(y,x))  # as this expression proves.
expression(y ~ x)
```

For developers, formula objects can be used to offer a more user-friendly interface, since they are closer to the human language. For example, the R formula `y∼x` can express that `y` and `x` are linked or that `y` is a function of `x`. Generally speaking, the developer bears the responsibility of interpreting the formula to perform the necessary tasks. This is very advanced; we refer the interested reader to the R documentation files. Here are a few examples with no particular meaning, but which will help become familiar with this new object:

```
> y~x
y ~ x
> y~(x+y:z)*t|v
y ~ (x + y:z) * t | v
> y1+y2|w ~ (x+y:z)*t|v
y1 + y2 | w ~ (x + y:z) *t | v
```

It is worth pointing out that even if the quantities mentioned in the formulae above are not existing R objects, no error is thrown. However, remember that a syntax error results in an error message:

```
> y~x+y)*t|v
Error : ')' not expected in "y~x+y)"
```

We now focus on usage of formulae in the R system. Since formulae are not common objects, the user may not realize that they are saved like any other R object.

```
> form <- y~x
> form
y ~ x
```

The two main uses are for plots and for statistics.

For plots, this is an alternative to what we introduced in Chap. 7.

```
> x <- runif(10)
> y <- runif(10)
> plot(x,y)
> plot(y~x)
```

The resulting plot is not shown here, since the only interest is in showing that the instructions with or without the formula are equivalent. Note that the variables `x` and `y` are inverted between the two forms. The version with the formula `plot(y~x)` expresses more literally the action we want: plot `y` *as a function of* `x`. This version, which we find elegant, is of course also available for the complementary functions `points()` and `lines()`.

In a statistical context, a function relative to the specific treatment of a statistical model takes as input argument a formula establishing the relationship between the variables of the model (the formula is often the first argument). The most simple example is the linear model; here is an example[11]:

```
> lm(y~x)    # x and y must be defined (and they are in this
            # case!)
Call:
lm(formula = y ~ x)
Coefficients:
(Intercept)            x
    0.46290     -0.06904
```

[11] This section does not give details on handling linear models in R; this will be the focus of Chap. 14.

```
> lm(form)  # Recall that: form <- y~x
Call:
lm(formula = form)
Coefficients:
(Intercept)             x
    0.46290      -0.06904
```

Besides the pleasant syntax, the formula object also offers a very efficient interface with the user to describe the model. This is confirmed by the fact that, unlike for plots, there is no other way of describing the relationship between the variables in the model. You might think that the syntax `lm(y,x)` could have been used. But then how would you write as a list of input arguments the formula `y~(x+z)*t` , which is perfectly valid (see Chap. 15)?

For operations on formulae, you can use the function `update()` which modifies a formula, using another one.

```
> update(y~x,.~.+z)      # Change y~x into y~x+z.
y ~ x + z
> form <- y~x            # The same procedure with a saved model.
> form2 <- update(form,.~.+z)
> form2
y ~ x + z
> update(form2,.~.-x)  # You can also delete a variable.
y ~ z
```

These examples show the syntax of the function `update()`. The first formal argument is the formula you wish to modify; the second formal argument gives the operations to perform on the formula, using a specific syntax. All that remains to be done is to analyse the syntax of the second formula. Any dot "." before the tilde character "~" is replaced with the left expression of the initial formula (before the tilde). Similarly, any dot "." after the tilde is replaced with the right expression of the initial formula (after the tilde).

8.4.2.3 The R Environment

The notion of environment is necessary in any programming language. It can be seen as a storage space of R objects. When you open your R session, a first environment `.GlobalEnv` is created by R. It is called the workspace and all objects manipulated with the command line during this session are stored there. Although we only wish to give an overview of this concept, it is worth mentioning that the notion of function depends intrinsically on the notion of environment. Here is a glimpse of this fact. When you create a new object in the body of a function, R takes care of declaring it internally in an environment specific to this function, to store the contents of the object. The reason for this is that if the object has the same name as an object of the environment `.GlobalEnv`, this last object will not be overwritten with the value defined in the body of the function. To better understand what an environment is, note that the value of an object defined in the environment `.GlobalEnv` can be accessed in the body of the function. However, its value cannot be modified by an affectation with the same object name. The reason why you can access an object which was defined in another environment than the one associated with the function is that a parent environment is specified when declaring a new environment. It is allowed that an environment has no parent, as is the case with the initial environment `.GlobalEnv`. When an object is not directly available in the environment of a function, R searches for the object in the parent environment. If it is still not available, there are two possibilities: either there exists a "grandparent" environment, and the search continues, or there is no such environment and an error is thrown indicating that the object could not be found. This exploration process is repeated recursively until the object is found. Most environment declarations are done internally and invisibly by R. We shall return to this notion when we give more details on developing functions. A very surprising feature is that an environment is considered as an R object. A new environment can thus be declared to execute a specific block of code without changing the workspace `.GlobalEnv`. The function `local()`, which takes as first argument the code to execute and as second argument the environment for the execution, is very useful to this end:

```
> a <- 12; b <- 13
> space <- new.env() # By default, the parent is the environment
                     # from which new.env is called.
> local({
+ a <- b+2
+ a
+ },space)
[1] 15
> a # The value of a has not changed in .GlobalEnv.
[1] 12
> space$a # Value of a in the environment space.
[1] 15
```

The function's name is well chosen: the value of a in the workspace `.GlobalEnv` has been preserved. As stated in the comment, the parent of `space` (generated by `new.env()`) is `.GlobalEnv`, but the parent could have been specified by giving a value to the formal argument `parent`. Here are two examples of parent declaration:

```
> space2 <- new.env(parent=emptyenv())
> local(a<-b+2,space2) # Error!!!
Error in eval(expr, envir, enclos) : could not find function "<-"
> space2$a   # Unsurprisingly, the object a does not exist!
NULL
```

The environment `space2` is useless, since its parent environment is an empty environment (i.e. no parent; declared with the function `emptyenv()`). The execution error in the local code is because even the affectation function <- cannot be accessed: the empty environment knows absolutely nothing about R; in particular, it does not know the basic functions. The function `globalenv()` returns the global environment `.GlobalEnv` which is always first in the access list of R environments.

```
> space3 <- new.env(parent=parent.env(globalenv()))
> local(a<-b+2,space3) # Error, because .GlobalEnv cannot be
                       # accessed!
Error in eval(expr, envir, enclos) : object 'b' not found
> local(a<-15,space3)
> a
[1] 12
> space3$a
[1] 15
```

Environments are rather convenient-they are used like a list. 585

```
> space3$b <- b-1
> b
[1] 13
> space3$b
[1] 12
```

For further details, we refer the reader to the online help, which is rather com- 586
plete, but aimed at advanced users. 587

SECTION 8.5

† Interfacing R and C/C++ or Fortran 588

You may be wondering why you should consider writing parts of your code in C/C++ 589
or Fortran. There are several reasons, such as: 590

- To use from R a pre-existing routine, formerly coded in C/C++ or Fortran 591
- To speed up the runtime of your R code 592
- To use the graphical capabilities of R or some R functions on numerical output 593
  from C/C++ or Fortran code 594

Tip

The last version of R includes a byte compiler which speeds up some computations. You can also use the R version distributed by the company Revolution Analytics (http://www.revolutionanalytics.com). It has been optimized to speed up some computations, for example, by relying on a multi-core architecture when available.

Warning

Interfacing R and C/C++ or Fortran is much more convenient under Linux (or MacOS) than under a Microsoft Windows OS for which several necessary tools lack. Note that the authors of this book use Linux on a daily basis!

See also

We assume that the reader already has some notions of C/C++ and/or Fortran programming. If that is not the case, the books [22, 38] for C and C++, and [9] for Fortran may be of use.

In this section, we do not claim exhaustivity. We shall only present a few simple 595
examples which illustrate the points made above. Along the way, we shall provide 596
some basics which we hope will allow you to get by on your own afterwards. 597

Warning

Before you start, you need to install C/C++ and Fortran compilers, since Microsoft Windows does not have any by default. The free software Rtools, containing several tools from the Linux world, has been created to this end. You can download it from http://cran.r-project.org/bin/windows/Rtools. Choose Full installation to build 32 or 64 bit R 2.14.2+ if you have a 64 bit processor. Tick the appropriate box when installing Rtools, so that the variable PATH is correctly configured. You also need to change the system environment variable Path so that it contains the path to the R installation folder (one way to find the path is to right-click on the R icon of the desktop, then choose properties). This will allow you to call R from an MS-DOS command window, as we shall mention later on. To do this, right-click on the Windows Desktop, select New/Shortcut, then enter the following instruction in the window that opens:
`control.exe sysdm.cpl,System,3`
Once this shortcut has been created on the desktop, double-click on it, and in the window that opens, click on Environment Variables... Change the value on the system variable Path to add **at the beginning** (using ; as separator) the path to the folder containing the R executable (which should look like C:\Program Files\R\R-3.1.0\bin\i386 or C:\Program Files\R\R-3.1.0\bin\x64) and the path to the folders of Rtools (which should look like C:\Rtools\bin and C:\Rtools\gcc-4.6.3\bin), if they are not already present.

8.5.1 Creating and Running a C/C++ or Fortran Function 598

The next example shows how to speed up a program by using C/C++ or Fortran. 599
The R function combn() is able to handle all combinations of a given number of 600
elements taken from a given vector. For example, this instruction generates all com- 601
binations of size 3 from the vector 1:5. 602

```
> combn(5,3)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1    1    1    1    2    2    2     3
[2,]    2    2    2    3    3    4    3    3    4     4
[3,]    3    4    5    4    5    5    4    5    5     5
```

If we attempt to get all `choose(n,m)` combinations (e.g., 1,313,400 combinations if $n = 200$ and $m = 3$) from a vector of larger size n, the computation time can increase drastically.

```
> system.time(x <- combn(200,3))
   user  system elapsed
 14.959   0.227  15.188
```

The command `system.time()` shows that the above computation takes several seconds on the computer used to write this book (if your computer is faster, take a value greater than 200).

Tip

The function `permn()` of package `combinat` can be used to generate all permutations of the elements of a vector.

A simplified version of the original R function `combn()` is given below:

```
> combnR <- function(n,m) {
+  a <- 1:m ; e <- 0 ; h <- m
+  combmat <- matrix(0,nrow=m,ncol=choose(n,m))
+  combmat[,1] <- 1:m
+  i <- 2
+  nmmp1 <- n - m + 1
+  mp1 <- m + 1
+  while (a[1] != nmmp1) {
+   if (e<n-h) {
+     h <- 1 ; e <- a[m] ; a[m-h+1] <- e + 1
+     combmat[,i] <- a
+     i <- i + 1
+  } else {
+    h <- h + 1 ; e <- a[mp1-h]
+    a[(m-h+1):m] <- e + 1:h
+    combmat[,i] <- a
+    i <- i + 1
+  }}
+ return(combmat)
+ }
```

We now propose two functions coded in C/C++, and another two coded in `Fortran`, to make the same computation in much shorter time.

• Creating the C/C++ function

C++ code for function `combnC`, downloadable from `http://biostatisticien.eu/springeR/combn.cpp`:

```
#include <math.h>
extern "C" {
void combnC(int *combmat, int *n, int *m) {
 int i, j, e, h, nmmp1, mp1;
 int *a;
 a=new int[m[0]];
 for (i=1;i<=m[0];i=i+1) a[i-1]=i;
 e=0;
 h=m[0];
 for (i=1;i<=*(m+0);i=i+1) combmat[i-1]=i;
 i=2;
 nmmp1=n[0] - m[0] + 1;
 mp1=m[0] + 1;
 while(a[0] != nmmp1) {
  if(e < n[0] - h) {
  h=1;
  e=a[m[0]-1];
  a[m[0] - h]=e + 1;
 for (j=1;j<=m[0];j=j+1) combmat[(i-1)*m[0]+j-1]=a[j-1];
  i=i+1;
  }  else {
  h=h + 1;
  e=a[mp1 - h-1];
  for (j=1;j<=h;j=j+1) a[m[0] - h + j-1]=e + j;
 for (j=1;j<=m[0];j=j+1) combmat[(i-1)*m[0]+j-1]=a[j-1];
  i=i + 1; }          }
 delete[] a;
}}
```

Code for the main function, downloadable from http://biostatisticien.eu/springeR/main.cpp:

```
#include <iostream>
using namespace std;
extern "C" {
int main() {
  void combnC(int *combmat, int *n, int *m);
  int *n, *m, *combmat, j;
  double Cnm;
  n=new int[1];
  m=new int[1];
  n[0]=5;
  m[0]=3;
  Cnm=10;
  combmat=new int[(int)Cnm*m[0]];
  combnC(combmat,n,m);
  for (j=1;j<=Cnm*m[0];j++) cout << combmat[j-1] << " ";
}}
```

Note that all indices start at zero in C/C++, unlike R where they start at 1.

- Creating the Fortran function

Fortran code for the subroutine combnF, downloadable from http://biostatisticien.eu/springeR/combn.f90:

```
SUBROUTINE  combnF(combmat,n,m)

integer, intent(in) :: n,m
integer             :: i,j,e,h,nmmp1,mp1
integer,dimension(m) :: a
integer,dimension(*), intent(out)::combmat

do  i=1,m
a(i)=i
end do
e=0
h=m
do  i=1,m
combmat(i)=i
end do
i=2
nmmp1=n-m+1
mp1=m+1
do while (a(1) .ne. nmmp1)
if (e < n-h) then
h=1
e=a(m)
a(m-h+1)=e+1
do  j=1,m
combmat((i-1)*m+j)=a(j)
end do
i=i+1
else
h=h+1
e=a(mp1-h)
do 40 j=1,h
a(m-h+j)=e+j
40 continue
do j=1,m
combmat((i-1)*m+j)=a(j)
end do
i=i+1
endif
enddo
END SUBROUTINE  combnF
```

Code for the main function, downloadable from http://biostatisticien.eu/springeR/main.f90:

```
PROGRAM main
integer :: n,m,Cnm,j,k
integer, allocatable,dimension(:) :: combmat
n=5
m=3
Cnm=10
k=Cnm*m
allocate(combmat(k))
CALL combnF(combmat,n,m)
write(*,*) (combmat(j) ,j=1,k)
deallocate(combmat)
end PROGRAM main
```

• Compiling and running the C/C++ or Fortran function

In order to use the C++ or Fortran code given above, it needs to be compiled, i.e. transformed into an executable file. To do this, simply open an MS-DOS terminal window, for example, from the Windows menu Start/Run (or with the keyboard combination [WINDOWS+R]) and type the instruction cmd followed by ENTER. In this black window, type the two instructions below.

Warning

You may need to move to the directory where your files were saved, using the MS-DOS command cd (for *change directory*). For example, if you created your files on the Windows Desktop, use

```
cd Desktop
```

Note that under MS-DOS, the command dir is used to list the contents of the current directory.

```
:: To compile C/C++ code:
g++ -o mycombn.exe combn.cpp main.cpp
:: To compile Fortran code:
gfortran -o mycombn.exe combn.f90 main.f90
:: To run the function:
mycombn.exe
```

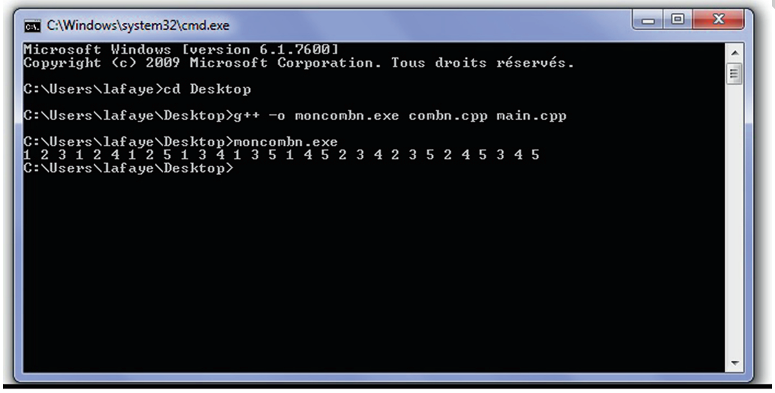
The first instruction compiles our C++ or Fortran code to produce the executable file mycombn.exe. The second instruction launches this executable file and displays, though with no formatting, the result of the computation.

Tip

The function `system()` is used to execute a DOS command outside of R. For example, in R, type:

```
> system("mycombn.exe")
1 2 3 1 2 4 1 2 5 1 3 4 1 3 5 1 4 5 2 3 4 2 3 5 2 4 5 3 4 5 >
```

Note that you must of course first change the current R directory, using function `setwd()`, for example, to change to the directory containing the file mycombn.exe.



```
Microsoft Windows [version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

C:\Users\lafaye>cd Desktop

C:\Users\lafaye\Desktop>g++ -o moncombn.exe combn.cpp main.cpp

C:\Users\lafaye\Desktop>moncombn.exe
1 2 3 1 2 4 1 2 5 1 3 4 1 3 5 1 4 5 2 3 4 2 3 5 2 4 5 3 4 5
C:\Users\lafaye\Desktop>
```

Tip

The compilation flag `-Wall` is used to display all compilation warnings or errors (if there are any!):

```
g++ -o mycombn.exe combn.cpp main.cpp -Wall
```

We shall now produce the $\binom{200}{3} = 1,313,400$ sub-vectors made of all possible combinations of three elements in vector `1:200`. For the C/C++ version, modify lines 11, 13 and 16 of the code of function `main` given p. 233. These lines become

```
n[0]=200;
Cnm=1313400;
// for (j=1;j<=Cnmm[0];j++) cout << combmat[j-1] << " ";
```

For the `Fortran` version, modify lines 4, 6 and 10 of the code of function `main` given p. 235. These lines become

```
n=200
Cnm=1313400
!write(*,*) (combmat(j) ,j=1,k)
```

We commented out the last line (using // in C/C++ and ! in Fortran) so that a call of mycombn.exe no longer displays the (now very large) result of the computation, which would take a lot of time. But the calculation is made. We are thus coherent with the previous computation done in R, for which the result was not displayed but stored in variable x. After saving your changes, recompile and run your code:

```
:: To compile C/C++ code:
g++ -o mycombn.exe combn.cpp main.cpp
:: To compile Fortran code:
gfortran -o mycombn.exe combn.f90 main.f90
:: Execute the function:
mycombn.exe
```

You can see that the calculation (without displaying the result) is done very quickly.

8.5.2 Calling C/C++ (or Fortran) from R

We shall now see how to call the C++ code from file combn.cpp (or rather a compiled version of this code) directly from R, without using a main function. To this end, we create an R wrapper containing a call of the C++ function.

Note

R can only call C/C++ or Fortran functions which do not return any output. All C/C++ functions must thus be of type void and all Fortran routines must be subroutines. The results will go in the arguments of the calling function.

Download the file http://biostatisticien.eu/springeR/combn.R, which includes the code given below:

```
combnRC <- function(n,m) {
 combmat <- matrix(0,nrow=m,ncol=choose(n,m))
 lib <- paste("combn",.Platform$dynlib.ext,sep="")
 dyn.load(lib)
 out <- .C("combnC",res=as.integer(combmat),
                    as.integer(n),as.integer(m))
 combmat <- matrix(out$res,nrow=m,byrow=F)
 dyn.unload(lib)
return(combmat)
}
```

Note

To call the Fortran code, replace line 5 by

```
out <- .Fortran("combnF",res=as.integer(combmat),
```

The functions `dyn.load()` and `dyn.unload()` allow respectively to load and unload from R's memory the resources from a DLL (dynamic link library) file. A DLL includes functions which can be called during the execution of a program, without being included in its executable. Here, it is the file combn.dll (which includes only one function), which will be created further on.

The functions `.C()` and `.Fortran()` (which output a list) are used to send values from R to a C/C++ or Fortran function, respectively. Use the instructions `as.integer()`, `as.double()` or `as.character()` in R to declare objects made of integer values, decimal (numeric) values or character strings, so that they are "received" correctly by the arguments of the C/C++ or Fortran function.

For a C/C++ function, all arguments must be pointers, for example, integer pointers (`int` *), real pointers (`double` *) or character pointer pointers (`char` **). Table 8.1 gives the equivalent types in R, C/C++ and Fortran.

Table 8.1: Conventions on argument types. Type `?.Fortran` for further detail

R	C/C++	Fortran
integer	int *	INTEGER
numeric	double *	DOUBLE PRECISION
numeric	float *	REAL
complex	Rcomplex *	DOUBLE COMPLEX
logical	int *	integer
character	char **	CHARACTER*255
list	SEXP *	not allowed
other type	SEXP	not allowed

Warning

Unlike R, where it is very easy to get the length of vector `x` with the instruction `length(x)`, in C/C++ it is not possible to know the length of `x`. It can sometimes be useful to give to the function `.C()` both the vector `x` and its length, for example, as follows for some hypothetical function `functionC`:

```
x <- c(1.2,0.7,3,2,4,1,0.9)
.C("functionC",as.double(x),as.integer(length(x)))
```

The arguments of the C/C++ function `functionC` are `double` *`x` and `int` *n. The same remark applies to Fortran functions.

Note

The C/C++ function combnC returns *void*: it does not have any direct output. However, the value of its arguments, which are pointers, can be modified during execution. It is then possible to access directly (thanks to their address) to the value of these pointers. This is how R works, using the function `.C()` (in a transparent way for the user).
You may have noted at line 5 of the code of function `combnRC()` above that we used `res=` when calling function `.C()`. This allows us to use `out$res` directly, instead of `out[[1]]`. You can use another name than `res`, and for any argument of function `.C()`. For example, we could have used `val=as.integer(m)`, which we did not do because that value was not modified by combnC and is thus already known (as `m`). A similar remark applies to Fortran functions.

We shall now create the file combn.dll, which will be called by R. To this end, type the following instructions in an MS-DOS window:

```
:: In C/C++:
g++ -c combn.cpp -o combn.o
g++ -shared -o combn.dll combn.o
:: In Fortran:
gfortran -c combn.f90 -o combn.o
g++ -shared -o combn.dll combn.o
```

Tip

Equivalently (or almost equivalently, since optimization arguments could be used by the compiler, which might by the way hinder debugging), this dynamical library could be created (after deleting if necessary the files combn.o and combn.dll) with one instruction:

```
R CMD SHLIB combn.cpp -o combn.dll
```

The first instruction creates the object file `combn.o`, which contains the machine code for the function included in file `combn.cpp`. The second instruction creates the dynamic library `combn.dll`. At this step, the compiler informs us of any errors to correct in the program (with the corresponding line number).

Tip

Note that it is possible to include several object files in the same library, which will then contain several functions. For example, if we had a file choose.o containing the machine code for a function which calculates binomial coefficients, we could include both functions in a DLL as follows:

```
g++ -shared -o combn.dll combn.o choose.o
```

Linux

Under Linux, DLL files usually have a `.so` extension (for *shared object*). You should thus replace all occurrences of extension `.dll` by extension `.so`.

Mac

Under MacOS, DLL files usually have a `.dylib` extension (for *dynamic library*). You should thus replace all occurrences of extension `.dll` by extension `.dylib`. Also note that under MacOS, you must replace `g++ -shared` with `g++ -dynamic`.

In R, after changing to the correct directory, we can now execute the following instructions: 818 819

```
> combn(5,3)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1    1    1    1    2    2    2     3
[2,]    2    2    2    3    3    4    3    3    4     4
[3,]    3    4    5    4    5    5    4    5    5     5
> source("combn.R")
> combnRC(5,3)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1    1    1    1    2    2    2     3
[2,]    2    2    2    3    3    4    3    3    4     4
[3,]    3    4    5    4    5    5    4    5    5     5
> system.time(x <- combn(200,3))
   user  system elapsed
 14.803   0.229  15.035
> system.time(x <- combnRC(200,3))
   user  system elapsed
  0.158   0.023   0.181
```

There is an important speed-up, thanks to this new R function using code written in C/C++. 820 821

Do it yourself

Code in R alone, and then in hybrid R-C/C++ (or R-Fortran), the functions `ar1simR()` and `ar1simRC()-ar1simC` (or `ar1simRF()-ar1simF`). These functions take three input arguments: $n \in \mathbb{N}$, $\phi \in (-1, 1)$ and $M \in \mathbb{N}$. They do the following computations.

For $m = 1, \ldots, M$:

(a) Simulate random vector $\boldsymbol{\epsilon} = (\epsilon_1, \ldots, \epsilon_n)^{\mathsf{T}}$ with distribution $\mathcal{N}_n(\mathbf{0}; \mathcal{I}_n)$.
(b) Create vector $\boldsymbol{x} = (x_1, \ldots, x_n)^{\mathsf{T}}$, with $x_1 = \epsilon_1$, and such that for all $t = 2, \ldots, n$, we have $x_t = \phi x_{t-1} + \epsilon_t$.
(c) Calculate the conditional least squares estimator $\hat{\phi}_m$ of ϕ:

$$\hat{\phi}_m = \frac{\sum_{t=2}^{n} x_{t-1} x_t}{\sum_{t=2}^{n} x_{t-1}^2}.$$

The functions you create should output the value $\overline{\hat{\phi}} = \frac{1}{M}\sum_{m=1}^{M} \hat{\phi}_m - \phi$, thus allowing a numerical evaluation of the bias of estimator $\hat{\phi}$ of ϕ.

Compare the speed of execution of the pure R version with the version calling C/C++ (or Fortran) code. To this end, plot the values $(M, time_M)$ for $M = 1{,}000, 2{,}000, \ldots, 100{,}000$. Take $n = 1{,}000$ and $\phi = 0.75$.

Note: The function `arima.sim()` performs parts (a) and (b) above, and function `arima()` performs part (c). Do not use these two pre-existing functions for this exercise: they are very fast because they are coded in C, but are not limited to the previous computations.

Tip

To ease code development, a good editor is always useful. An editor should at least include indentation and syntactical colouring. You may wish to use the following free software:

- An R code editor such as `RStudio`, `Tinn-R` or `Emacs`
- A source code editor for C/C++ and `Fortran` such as `Emacs` or `Code::Blocks` (available at `http://www.codeblocks.org`)

Tip

The package rbenchmark can be used to easily calculate the expected gain in computation time by using an R-C/C++ or R-Fortran function rather than a pure R function. For example, try to verify the results we got in the previous practical using the following code:

```
n <- 1000
phi <- 0.75
M <- 2000
dyn.load("ar1sim.dll")
benchmark(Rcode=ar1simR(n,phi,M),
          Ccode=.C("ar1simC",as.integer(n),phi,
                   as.integer(M),res=0.0)$res,
          replications=1000)
```

Tip

Fortran and R store matrices (tables) in the same way: the rows of a given column are stored sequentially in memory. In C/C++, the opposite holds; columns of a given line are stored sequentially. Be careful when sending a matrix from R to C/C++. For example, the element with index [i,j] in an R matrix corresponds to the element with index [(j-1)*number-of-rows + (i-1)] in C/C++ (in C/C++, indices start at 0).

8.5.3 *Calling External C/C++ or Fortran Libraries* 849

It is possible to use a function from an **external** library, thanks to the R functions 850
.C() (for C/C++ libraries) and .Fortran() (for Fortran libraries). 851

Tip

Here is an amusing application of this approach, which locks the Windows session:

```
# Select file C:/windows/system32/user32.dll:
dyn.load(file.choose())
.C("LockWorkStation")
```

It is also possible to call an external library directly from your C/C++ or Fortran 852
code. Here are some scientific libraries which we find interesting: 853

- The R API (*application programming interface*)
- The C++ library newmat
- The Fortran libraries BLAS and LAPACK

See also

Other libraries exist; some are free of charge, or even open-source, such as:

- In C/C++:
  - http://www.gnu.org/software/gsl
  - http://www.math.uiowa.edu/~dstewart/meschach
  - http://www.nrbook.com/a/bookcpdf.php
- In Fortran:
  - http://calgo.acm.org
  - http://www.nrbook.com/a/bookfpdf.php
  - http://www.nrbook.com/a/bookf90pdf.php
  - http://math-atlas.sourceforge.net

Others are not free:

- In C/C++:
  - http://www.nag.co.uk/numeric/CL/CLdescription.asp
  - http://www.vni.com/products/imsl/c/imslc.php
- In Fortran:
  - http://www.nag.co.uk/numeric/RunderWindows.asp
  - http://www.nag.co.uk/numeric/fl/FLdescription.asp
  - http://www.nag.co.uk/numeric/fn/FNdescription.asp
  - http://www.vni.com/products/imsl/fortran/overview.php

8.5.3.1 The R API

The R API is a library created by the R developers. It can be used from a C/C++ program without even using R (it is then called standalone R API). It can also be used in C/C++ code which will itself be called from R, as introduced in the previous section. This allows the use of existing routines without having to rewrite them. To use this library, you must include in your C/C++ source code the two header files R.h and Rmath.h, which are necessary to declare or define some mathematical functions and constants.

See also

The documentation for this library, which includes the list of functions and constants contained in it, is available at `http://cran.r-project.org/doc/manuals/R-exts.html#The-R-API`.
You may also find interesting to consult the contents of the directory `nmath/` in the R sources; it is available at `http://svn.r-project.org/R/trunk/src/nmath`.

We present below C/C++ code available at `http://biostatisticien.eu/springeR/integ.cpp` which allows to compute the integral

$$\int_{\epsilon_1}^{\pi} \Phi(t+\epsilon_2)dt,$$

where ϵ_1 and ϵ_2 are realizations of two independent random variables (respectively, normal and uniform) and where $\Phi(\cdot)$ is the cumulative distribution function of the $\mathcal{N}(0,1)$ distribution. The only point of this example is to illustrate the use of the R API to simulate random variables, calculate a probability and perform numerical integration.

```
#include <R.h>
#include <Rmath.h>

extern "C" {

  typedef void integr_fn(double *x, int n, void *ex);
  void f(double *t, int n, void *ex);
  void testintegral(double *res) {

   // R API numerical integration function
   void Rdqags(integr_fn f,void *ex,double *a,
               double *b,double *epsabs,
               double *epsrel,double *result,
               double *abserr,int *neval,
               int *ier,int *limit,int *lenw,
               int *last,int *iwork, double *work);

    GetRNGstate(); // Read the R generator seed

    double *a, *b, *epsabs, *epsrel, *result,
      *ex, *abserr, *work;
    int *last, *limit, *lenw, *ier, *neval, *iwork;

    ex = new double[1]; a = new double[1];
    b = new double[1]; epsabs = new double[1];
  epsrel = new double[1]; result = new double[1];
  abserr = new double[1]; neval = new int[1];
  ier = new int[1]; limit = new int[1];
  lenw = new int[1]; last = new int[1];
  limit[0] = 100;
  lenw[0] = 4 * limit[0];
  iwork =  new int[limit[0]];
  work = new double[lenw[0]];

  a[0] = rnorm(0.0,1.0);   // eps1 from N(0,1)
  b[0] = M_PI; // The constant \pi (3.141593...)
  ex[0] = runif(0.0,1.0); // eps2 from Unif(0,1)

  // Calculate the integral
  Rdqags(f, ex, a, b, epsabs, epsrel,
         result, abserr, neval, ier,
         limit, lenw, last,
         iwork, work);

  // The result is stored in res[0]
  res[0] = result[0];

  PutRNGstate(); // Write the generator seed

  // Free up some memory
  delete[] ex, a, b, epsabs, epsrel, result, abserr,
    neval, ier, limit, lenw, last, iwork, work;
 }

 // Define the function to integrate
 void f(double *t, int n, void *ex) {
  int i;
  double eps2;
  eps2 = ((double*)ex)[0];
  for (i=0;i<n;i++) {
    t[i] = pnorm(t[i]+eps2,0.0,1.0,1,0);}
 }

}
```

The instructions to compile this function in order to get a DLL file are

```
g++ -c integ.cpp -o integ.o -I"C:\Program Files\R\R-3.1.0
     \include"
g++ -shared -o integ.dll integ.o ^
     -L"C:\Program Files\R\R-3.1.0\bin\i386" -lR
```

Warning

Note that we had to indicate the paths to the folders containing the files R.h, Rmath.h and R.dll. Modify these as needed depending on your system configuration. In MS-DOS, the symbol ˆ indicates an incomplete line.

Linux

```
g++ -c integ.cpp -o integ.o  -I"/usr/lib/R/include" -fPIC
g++ -shared -o integ.so integ.o -I"/usr/lib/R/include" \
        -L"/usr/lib" -lR
```

Now, to perform the calculation in R, use the following instructions: 944

```
> dyn.load(paste("integ",.Platform$dynlib.ext,sep=""))
> # i.e. dyn.load("integ.dll") under Windows.
> .C("testintegral",val=0.0)$val
[1] 3.707762
```

Of course, the result of this computation varies, depending on the realizations of 945
ϵ_1 and ϵ_2. 946

8.5.3.2 The newmat Library 947

The newmat library is used to manipulate various types of matrices and to 948
perform classical operations such as multiplication, transposition, inversion, eigen- 949
value computation and decompositions. 950

See also

The complete documentation for this library is available at http://www.robertnz.net/nm11.htm.

The code below, available at http://biostatisticien.eu/springeR/inv. 951
cpp, is C/C++ code using this library to invert a matrix and can be called from R. 952

```
#define WANT_STREAM
#define WANT_MATH
#include "newmatap.h"
#include "newmatio.h"
#ifdef use_namespace
using namespace NEWMAT;
#endif
extern "C" {
  void invC(double *values, int *nrow) {
      int i, j;
      Matrix M(nrow[0],nrow[0]);
     M << values;
     M << M.i(); // Calcul de l'inverse de M
      for (i=1;i<=nrow[0];i++) {
         for(j=1;j<=nrow[0];j++) {
            values[nrow[0]*(i-1)+j-1] = M(i,j);
            }
      }
     M.Release();
      return;
  }
}
```

Tip

Download file http://www.robertnz.net/ftp/newmat11.zip and unzip it in C:/newmat. Then type the following instructions in an MS-DOS window:

```
cd \
cd newmat
g++ -O2 -c *.cpp
ar cr newmat.a *.o
ranlib newmat.a
cp newmat.a newmat.dll
```

After a few minutes, the libraries newmat.a and newmat.dll are created in folder C:\newmat.

You now need to create the library inv.dll (or inv.so under Linux) using the following instructions:

```
cd folder containing file inv.cpp
g++ -IC:\newmat -o inv.o -c inv.cpp
R CMD SHLIB inv.cpp -IC:\newmat C:/newmat/newmat.a
```

Linux

```
g++ -I/usr/include/R  -I/usr/local/include -Inewmat -fpic \
    -c inv.cpp -o inv.o
R CMD SHLIB inv.cpp -Inewmat newmat/newmat.a
```

You can then use the C/C++ above from R as follows. First save the following code in a file called inv.R:

```
> inv <- function(M) {
+  n <- nrow(M)
+  return(matrix(.C("invC",Minv=as.vector(M),n)$Minv,
+ nrow=n,ncol=n))}
```

Then execute the instructions: 984

```
> dyn.load(paste("inv",.Platform$dynlib.ext,sep=""))
> A <- matrix(rnorm(9),nrow=3)
> solve(A) # The R function solve() inverts a matrix.
            [,1]        [,2]      [,3]
[1,] -0.09893572  0.04676191  1.155500
[2,] -0.47035376  1.10728717 -2.979609
[3,]  0.03415044 -1.07683806  1.456918
> inv(A)
            [,1]        [,2]      [,3]
[1,] -0.09893572  0.04676191  1.155500
[2,] -0.47035376  1.10728717 -2.979609
[3,]  0.03415044 -1.07683806  1.456918
```

The two functions `solve()` and `inv()` thus give the same result for matrix 985
inversion. As you can see, the speed-up for this operation is substantial. 986

```
> benchmark(Rcode=solve(A),Ccode=inv(A),replications=10000)
   test replications elapsed relative user.self sys.self
2 Ccode        10000   0.255 1.000000     0.256    0.000
1 Rcode        10000   1.378 5.403922     1.351    0.025
  user.child sys.child
2          0         0
1          0         0
```

8.5.3.3 The BLAS and LAPACK Packages 987

The BLAS (*Basic Linear Algebra Subprograms*) and LAPACK (*Linear Algebra PACK-* 988
age) packages are `Fortran` packages which perform many matrix operations. We 989
shall see how to use them on a simple example. 990

991

First download the archiver software `7-zip` available at `http://www.7-zip.` 992
`org/download.html`. Use this software (twice) to unzip (in two steps) the file 993
`http://www.netlib.org/lapack/lapack.tgz`. All files and subfolders (`BLAS`, 994
`CMAKE`, etc.) should be placed directly in a folder called `C:\lapack`. For example, 995
this folder will contain at its root a file called `make.inc.example`, which you must 996
rename to `make.inc` after changing the line `SHELL = /bin/sh` to `SHELL = sh`. 997
Then type the following instructions in an `MS-DOS` window: 998

```
cd C:\lapack                 999
make lapacklib blaslib       1000
```

After several minutes, the static packages `librefblas.a` and `liblapack.a` are 1001
created. 1002

See also

The documentation for these packages can be read at http://www.netlib.org/lapack/lug. It is also useful to read the source code of all BLAS and LAPACK routines you wish to use, as they contain a detailed description of the arguments the routines take.

Here is the Fortran code, also available at http://biostatisticien.eu/springeR/inv.f90, for a subroutine which computes the inverse of a matrix. It uses the subroutines external DGETRF and DGETRI from the Lapack package.

```
! Returns the inverse of a matrix calculated by finding
! the LU decomposition.  Depends on LAPACK.
subroutine invF(A,Ainv,m)
  double precision, dimension(m,m), intent(in) :: A
  double precision, dimension(size(A,1),size(A,2)), &
                         intent(inout) :: Ainv

  ! work array for LAPACK
  double precision, dimension(size(A,1)) :: work
  integer, dimension(size(A,1)) :: ipiv ! pivot indices
  integer :: n, info, m

  ! External procedures defined in LAPACK
  external DGETRF
  external DGETRI

  ! Store A in Ainv to prevent it from
  ! being overwritten by LAPACK
  Ainv = A
  n = size(A,1)

  ! DGETRF computes an LU factorization of
  ! a general M-by-N matrix A using partial
  ! pivoting with row interchanges.
  call DGETRF(n, n, Ainv, n, ipiv, info)

  if (info /= 0) then
     stop 'Matrix is numerically singular!'
  end if

  ! DGETRI computes the inverse of a matrix using
  ! the LU factorization computed by DGETRF.
  call DGETRI(n, Ainv, n, ipiv, work, n, info)

  if (info /= 0) then
     stop 'Matrix inversion failed!'
  end if
end subroutine invF
```

To compile this code, execute the following instructions from an MS-DOS window:

```
cd %HOMEPATH%/Desktop # To be changed to suit your needs.
gfortran -c inv.f90 -o inv.o -I"C:/lapack"
gfortran -shared -o inv.dll inv.o -I"C:/lapack" ^
     C:/lapack/liblapack.a C:/lapack/librefblas.a
```

Linux

Under Linux, use the following instructions:

```
gfortran -c inv.f90 -o inv.o -fPIC
gfortran -shared -o inv.so inv.o /usr/lib64/liblapack.so.3
```

After creating the file inv.dll (or inv.so under Linux) with the previous instructions, you can start R and type the following instructions:

```
> dyn.load(paste("inv",.Platform$dynlib.ext,sep=""))
> A <- matrix(rnorm(4),nrow=2)
> B <- matrix(0,nrow=2,ncol=2)
> .Fortran("invF",A,res=B,2L)$res
           [,1]       [,2]
[1,] -1.1812737  1.9822527
[2,] -0.1681507 -0.7224351
> solve(A)
           [,1]       [,2]
[1,] -1.1812737  1.9822527
[2,] -0.1681507 -0.7224351
```

8.5.3.4 Mixing C/C++ and Fortran Packages

It is possible to call C/C++ functions from Fortran code, thanks to the instruction F77_SUB(name). We illustrate this point in the next example, which generates two independent observations: one from a $\mathcal{N}(0,1)$ distribution and the other from the uniform distribution. The Fortran code below uses the C functions GetRNGstate, PutRNGstate, rnorm and runif from the R API, which we have already used in Sect. 8.5.3.1. Save it in a file called random.f.

```
      SUBROUTINE random(x,y)
       real*8 normrnd, unifrnd, x, y
      CALL rndstart()
       x = normrnd()
      y = unifrnd()
      CALL rndend()
      RETURN
      END
```

Then create the file `random.c` containing

```
#include <R.h>
#include <Rmath.h>
void F77_SUB(rndstart)(void) {GetRNGstate();}
void F77_SUB(rndend)(void) {PutRNGstate();}
double F77_SUB(normrnd)(void) {return rnorm(0,1);}
double F77_SUB(unifrnd)(void) {return runif(0,1);}
```

To create your DLL file, compile using the instructions

```
gfortran -c random.f -o randomf.o
gcc -c random.c -o randomc.o  -I"C:\Program Files\R\R-3.1.0
\include"gfortran -shared randomf.o randomc.o -o random.dll ^
      -L"C:\Program Files\R\R-3.1.0\bin\i386" -lR
```

Linux

Under Linux, use

```
gfortran -c random.f -o randomf.o -fPIC
gcc -c random.c -o randomc.o -I"/usr/lib/R/include" -fPIC
gfortran -shared randomf.o randomc.o -o random.so
```

You can now call your code from R using the instructions:

```
> dyn.load(paste("random",.Platform$dynlib.ext,sep=""))
> .Fortran("random", as.double(1), as.double(1))
[[1]]
[1] 1.542474
[[2]]
[1] 0.59143
```

It is also possible to call `Fortran` functions from `C/C++` code, using the following instructions:

`F77_NAME(name)` to declare a `Fortran` routine in `C`
`F77_CALL(name)` to call a `Fortran` routine from `C`
`F77_COMDECL(name)` to declare a `COMMON FORTRAN` block in `C`
`F77_COM(name)` to access a `COMMON FORTRAN` block from `C`

Here is a small example (with `Fortran77` for a change). Save the code below in a file called `combnCF.cpp`:

```
#include <R.h>
#include <Rmath.h>
extern "C" {
void combnCF(int *combmat, int *n, int *m) {
// Caution! No upper case in the name of the subroutine.
void  F77_NAME(combnf)(int *combmat, int *n, int *m);
F77_CALL(combnf)(combmat,n,m);
 }
}
```

Then type the following instructions in an MS-DOS command window to create the package which will be called from R:

```
g++ -c combnCF.cpp -o combnCF.o -I"C:\Program Files\R
\R-3.1.0\include"gfortran -c combn.f90 -o combn.o
g++ -shared -o combnCF.dll combnCF.o combn.o ^
     -L"C:\Program Files\R\R-3.1.0\bin\i386" -lR
```

Linux

Under Linux

```
g++ -c combnCF.cpp -o combnCF.o-I"/usr/lib/R/include"-fPIC
gfortran -c combn.f90 -o combn.o -fPIC
g++ -shared -o combnCF.so combnCF.o combn.o \
 -I"/usr/lib/R/include" -L"/usr/lib" -lR
```

Now modify the code of function combnRC() given p. 237:

- Change the name of this function to combnRCF().
- Replace "combn" and "combnC" with "combnCF".

Save this new code in a file called combnCF.R. Then type the following instructions in the R console:

```
> source("combnCF.R")
> combnRCF(5,3)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1    1    1    1    2    2    2     3
[2,]    2    2    2    3    3    4    3    3    4     4
[3,]    3    4    5    4    5    5    4    5    5     5
```

8.5.4 *Calling* R *Code from a C/C++ Program Called by* R

We have seen how to call a C/C++ (or Fortran) routine from R. It is also possible to use a type of pointer called SEXP (for *Simple EXPression*) and the function

.Call(). In this subsection, we only give a simple example. The reader can use this as inspiration for more complex examples.

See also

We refer the reader to the website http://cran.r-project.org/doc/manuals/R-exts.html#Handling-R-objects-in-C.

In the following example, we shall see how to call function pmvt() of package mvtnorm from C/C++ code, itself called from R. The function pmvt() computes the probability that a random vector following a multivariate Student distribution belongs to a specified hyperrectangle in $\mathbb{R}^n$.

Unlike the examples in the previous sections, which used the function .C(), we shall need the function .Call(). Furthermore, our C/C++ code will have to be a function (which we call pmvtC in the following) which returns a structure of type SEXP and which also takes arguments of type SEXP. The code below, available from http://biostatisticien.eu/springeR/pmvt.cpp, will be transformed into a DLL file and then called by the function .Call().

```
#include <R.h>
#include <Rdefines.h>
#include "Rmath.h"
#include <R_ext/Rdynload.h>
extern "C" {
  SEXP pmvtCR(SEXP Rupper,SEXP Rcorr,SEXP Rdf,
              SEXP Rpmvt,SEXP Renv,SEXP Rres) {
    SEXP R_fcall;
    if(!isFunction(Rpmvt) & (Rpmvt != R_NilValue))
            error("Rpmvt must be a function");
    if(!isEnvironment(Renv))
            error("Renv must be an environment");
    PROTECT(R_fcall = lang4(Rpmvt,Rupper,Rcorr,Rdf));
    REAL(Rres)[0] = REAL(eval(R_fcall , Renv))[0];
    UNPROTECT(1);
    return(Rres);
  }
}
```

To compile this file, use the following instructions:

```
g++ -c pmvt.cpp -o pmvt.o -I"C:\Program Files\R\R-3.1.0
     \include"
g++ -shared -o pmvt.dll pmvt.o ^
     -L"C:\Program Files\R\R-3.1.0\bin\i386" -lR
```

Linux

Under Linux, use the instructions

```
g++ -m64 -I/usr/include/R  -I/usr/local/include -fpic  \
       -c pmvt.cpp -o pmvt.o
R CMD SHLIB pmvt.cpp
# or:
g++ -m64 -shared -L/usr/local/lib64 -o pmvt.so pmvt.o \
       -L/usr/lib64/R/lib -lR
```

You can now call this function from R. First download the file 1158
http://biostatisticien.eu/springeR/pmvt.R which contains the following 1159
code: 1160

```
> pmvtRCR <- function(upper,corr,df) {
+  res <- 0.0
+  Rpmvt <- function(upper,corr,df) {
+    d <- length(upper)
+    pmvt(lower=rep(-Inf,d),upper=upper,delta=rep(0,d),
+    corr=matrix(corr,ncol=d),df=df)}
+  dyn.load(paste("pmvt",.Platform$dynlib.ext,sep=""))
+  res <- .Call("pmvtCR",as.double(upper), as.double(corr),
+                as.double(df),Rpmvt,new.env(),as.double(res))
+  dyn.unload(paste("pmvt",.Platform$dynlib.ext,sep=""))
+ return(res)
+ }
```

Then type the following instructions: 1161

```
> require("mvtnorm")
> corr <- diag(3)
> set.seed(1)
> source("pmvt.R")
> pmvtRCR(c(2,3,2),corr,c(1,1,1))
[1] 0.706062
> set.seed(1)
> pmvt(lower=rep(-Inf,3),upper=c(2,3,2),corr=corr,df=c(1,1,1))[1]
[1] 0.706062
```

Tip

If an SEXP object contains a vector (e.g., SEXP x) or a matrix (e.g., SEXP M), you can use the instructions R_len_t n = length(x) and R_len_t p = nrows(M) to create integers containing the length n of vector x or the number of rows p of matrix M. The file Rinternals.h contains the list of many similar useful functions.

8.5.5 *Calling* R *Code from* `Fortran`

We recommend the open-source software `RFortran` available at `http://www.rfortran.org`.

8.5.6 *Some Useful Functions*

Here are a few functions which you may find useful. The following functions are used in an `MS-DOS` terminal window (or in `Cygwin`, see p. 258):

- `nm`: list of symbols of object files (e.g., `nm random.dll`).
- `objdump`: information about object files (e.g., `objdump -x random.dll`).
- `ldd`: list dynamic dependencies if necessary (e.g., `ldd random.dll`).

The following functions are used in R:

- `getLoadDLLs()`: list all DLLs loaded in the current session (e.g., `getLoadDLLs()`)
- `is.loaded()`: checks whether a library is loaded (e.g., `is.loaded(random.dll)`)

SECTION 8.6

† Debugging Functions

In this section, we present various options which can be useful to debug a function and find an error. The error could be either in the R code or in C/C++ or `Fortran` code called from your R function.

See also

We refer the reader to the website `http://www.stats.uwo.ca/faculty/murdoch/software/debuggingR`.

8.6.1 *Debugging Functions in Pure* R

We present some debugging functions, useful when writing R code.

The Function `browser()`

A useful debugging function in R is the function `browser()`. If you insert the instruction `browser()` in the source of your function, the program will stop at the place where it was inserted.

Here is an example showing how to use `browser()` in a function called `lsq()` which calculates the least squares estimator of unknown arguments in a simple linear model (see Chap. 14 for further details).

```
lsq <- function(X,Y,intercept=TRUE){
   X <- as.matrix(X)
   Y <- as.matrix(Y)
    plot(X,Y)
    nbunits <- nrow(X)
browser()
    if (intercept==TRUE) X <- cbind(rep(1,nbunits),X)
    betahat <- solve(t(X)%*%X)%*%t(X)%*%Y
    curve(betahat[1]+betahat[2]*x,add=TRUE)

return(betahat)
}
```

Source the file containing the previous code (e.g., with the instruction `source(file.choose())`), then type:

```
lsq(X=cars[,2],Y=cars[,1])
```

As you can see, the program stops and you can examine the contents of all local variables defined before `browser()`. For example, type `nbunits`.

Note

By typing the letter `n` (for *next*), you can inspect the code and the contents of variables sequentially. To leave the inspection mode, type `Q`.

Here is an overview of a debugging session:

```
lsq(X=cars[,2],Y=cars[,1])
Called from: mc(X = cars[, 2], Y = cars[, 1])
Browse[1]>nbunits
[1] 50
Browse[1]> betahat
Error: Object "betahat" not found
Browse[1]> n
debug: if (intercept == T) X <- cbind(rep(1, nbunits), X)
Browse[1]> n
debug: betahat <- solve(t(X) %*% X) %*% t(X) %*% Y
Browse[1]> n
debug: curve(betahat[1] + betahat[2] * x, add = T)
Browse[1]> betahat
          [,1]
[1,] 8.2839056
[2,] 0.1655676
Browse[1]> Q
>
```

Note

If you enter the letter `c` (for *continue*), the code is executed until the end, unless a `browser()` command is met again.

The Function `debug()`

Another interesting function is `debug()` which is equivalent to putting the instruction `browser()` at the top of a function. Thus `debug(var)` marks the functions `var` as debuggable. Any subsequent call of this function will launch the online debugger.

```
debug(var)
var(1:3)
```

To get rid of this mark, use the function `undebug()`.

```
undebug(var)
```

8.6.2 *Error in* R *Code*

First change line 6 of file combn.R, replacing the affectation arrow <- by the symbol <. We now have an error: an omitted symbol (the symbol -):

```
combmat<matrix(out$res,nrow=m,byrow=F)
```

Save the file, source it and type the following instruction:

```
> combnRC(5,3)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    0    0    0    0    0    0    0    0    0     0
[2,]    0    0    0    0    0    0    0    0    0     0
[3,]    0    0    0    0    0    0    0    0    0     0
```

As you can see, there is an error in the result, and the error that we introduced deliberately in the code could be difficult to detect if it were an accidental omission. Here is how we could try to detect where the error comes from. First install and load the package debug. Then use the function `mtrace()` of this package, as follows:

```
mtrace(combnRC)
combnRC(5,3)
```

You should then see a debugging window with the source code of function `combnRC()`. Pressing the `RETURN` key repeatedly will evaluate your source code line by line until the next display (which occurs upon evaluation of the line we modified):

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

This hints that there is an issue at this point. We can then correct the error, for example, with the instruction `fix(combnRC)`.

Note that the function `mtrace()` did not allow us to delve into the details of the following call:

```
.C("combnC",res=as.integer(combmat),as.integer(n),
                  as.integer(m))
```

8.6.3 *Error in the C/C++ or Fortran Code*

We shall now see how to perform the same kind of debugging for parts of the code written in C/C++ or Fortran. It mostly boils down to using the compilation option -g to add information on the source code in the DLL file, and then to using a specialized debugging tool.

Warning

You will need a debugging tool. We recommend the free software GDB. Download version 7.4 (32 bits) from `http://biostatisticien.eu/springeR/32/gdb.exe` and put it in the folder `C:\Rtools\bin`. This software uses the command line and is rather austere. You may find useful to add a graphical user interface (GUI), such as the Data Display Debugger (DDD) or Emacs. Under Windows, another interesting avenue is the software Insight, included in the set of tools MinGW, available from `http://sourceforge.net/projects/mingw/files/OldFiles/insight.exe/download`. However, this software seems to be becoming obsolescent. If you try to use it, remember to change the system environment variable Path to add the path to Insight (probably `C:\insight\bin`), as explained p. 231.
Under Microsoft Windows, you will have to install the version of Emacs available at `http://vgoulet.act.ulaval.ca/en/emacs/windows`. It is a bit more complicated to use DDD under Windows. You need to launch the Cygwin

setup (available at `http://cygwin.com/install.html`), choose the installation directory `C:\Rtools\bin` and select the software `Devel: ddd` and `Math: gnuplot` (and accept the required dependencies). Also note that if the list of download sites is empty, you can try the URL `http://cygwin.mirrorcatalogs.com`. To use DDD, you also need an implementation of the Linux `X` window system for Microsoft Windows. The software `Xming`, available at `http://biostatisticien.eu/springeR/Xming-6-9-0-31-setup.exe`, is a good choice. You could also use `MobaXterm` (`http://mobaxterm.mobatek.net`), or Cygwin's `Xorg` server (select `X11: xorg-server: X.Org servers` on installation).

8.6.4 Debugging with GDB

Start an `MS-DOS` command window from the Windows Start menu (type `cmd`) in which you type

```
cd path to folder containing inv.cpp
g++ -IC:\newmat -o inv.o -c inv.cpp -g
g++ -shared -o inv.dll inv.o -IC:\newmat C:/newmat/newmat.a
```

This will create the file `inv.dll` with debugging information (see p. 247 for the creation of the library `newmat`).

Tip

In order to also debug the functions from library `newmat`, you need to first create this library in a way that includes debugging information:

```
cd \
cd newmat
g++ -c *.cpp -Wno-deprecated -g
ar cr newmatdebug.a *.o
ranlib newmatdebug.a
cp newmatdebug.a newmatdebug.dll
```

Then type:

```
gdb Rgui
(gdb) run
```

This should start R, where you type

```
> setwd("path to file inv.dll")
> dyn.load("inv.dll")
```

Then go to menu Misc/Break to debugger, which will allow you to return to GDB (black window), where you can type

```
(gdb) info share
(gdb) break inv.cpp:1
(gdb) signal 0
```

The first instruction (info share) shows that the library inv.dll has been loaded; the second instruction (break inv.cpp:1) allows you to add a break point on the first (executable) line of the file inv.cpp; the last instruction (signal 0) exits GDB and returns to R. In R, type:

```
>  A <- matrix(rnorm(4),nrow=2)
>  source("inv.R") # File created page 247.
>  inv(A)
```

When the processor encounters the break point, the code execution is suspended. You can now type the following instructions in GDB. The first instruction (list) displays the next lines to execute, the second instruction (next) moves to the next line, the third instruction (print nrow[0]) displays the value of nrow[0] and the last instruction continues the code execution until the end or the next break point.

```
(gdb) list
(gdb) next
(gdb) print nrow[0]
(gdb) continue
```

You are back in R and you see the output of the call inv(A). You can type the following instructions to verify that the result is the same as with function solve() and to exit R.

```
>  solve(A)
>  q()
```

Linux

Under Linux, type in a terminal window the command

```
R -d gdb
```

instead of gdb Rgui.
Alternatively, you could use the following instructions:

```
export R_HOME=/usr/lib64/R
gdb /usr/lib64/R/bin/exec/R
```

To return to R from GDB, use the key combination CTRL+C. Note that to go from GDB to R, after typing signal 0 (or equivalently c), you need to press RETURN.

Tip

Note that GDB can be called with options. For example,

```
--directory=DIR    Search for source files in DIR.
--pid=PID          Attach to running process PID.
```

See also

The documentation of GDB, available at http://sourceware.org/gdb/current/onlinedocs/gdb, is worth reading.

Tip

You can install/compile a package (hereafter called PKG) with debugging information (equivalent to using the flag -g mentioned above). First create a file called Makevars.win (Makevars under Linux) in a subfolder called .R/ in your %HOME% directory. This file should include the following lines:

```
## for C++ code
CXXFLAGS=-g
```

For this purpose, you can for example type WINDOWS+R, cmd, ENTER, cd %HOME%, ENTER, mkdir .R, ENTER, cd .R, ENTER, echo CXXFLAGS=-g > Makevars.win, ENTER. Next, build the package PKG and install it (from the sources using the command R CMD INSTALL --build --debug PKG), then use one of the debugging methods presented above. Note that the file NAMESPACE of your package PKG must include the line useDynLib("PKG") so that the DLL (or .so) file is automatically loaded when you execute in R the instruction require("PKG"). If this procedure fails, you can always use the function dyn.load() to load the package "by hand" from where it is installed.

Tip

It is also possible to display the contents of an object of type SEXP (call this object s). To do this, you can include in your C/C++ code the instruction PrintValue(s);. This way, when the instruction is encountered during code execution, the contents of the object s will be displayed in the R console. Another solution is to use the instruction p Rf_PrintValue(s) from the GDB console. Note that in this case, the display of object s in the R console may be delayed until R takes over from GDB.

8.6.4.1 Debugging with Emacs

We have seen how to debug code with GDB. We shall now show how to perform the same kind of operations with the combination of Emacs (and its excellent module ESS, *Emacs Speaks Statistics*) and GDB. Note that you need to have installed GDB as explained in Sect. 8.6.3. Note also that you need to create, from an MS-Dos window, the file combn.dll with debugging information (flag -g), thanks to the following instructions:

```
g++ -g -c combn.cpp -o combn.o
g++ -shared -o combn.dll combn.o
```

Note

Under Emacs, the notation M-x means you must press simultaneously the keys ALT and X, whereas C-x means you must press simultaneously the keys CTRL and X, and [RET] designates the carriage return (key RETURN).

First open Emacs (see p. 258 for how to install this software) then execute the following commands. For example, the first line is executed by pressing simultaneously on ALT and X, then R (which will display M-x R at the bottom of Emacs), then RETURN (which will display ESS [S(R): R (newest)] starting data directory? ~/), then RETURN again (which will start R in Emacs).

```
M-x R [RET] [RET]
M-x gdb [RET] gdb -i=mi --annotate=3 [RET]
```

Your Emacs window should then be split in two, with R on top and GDB at the bottom. If that is not the case, go to the menu File/Split Window or File/New Window Below (C-x 2), then to the menu Buffer to select *R* *.

Warning

The system environment variable Path must include the entry C:\Rtools\bin first, so that the version of GDB used is 7.4.

You then need the process ID of R. Under Windows, use the key combination CTRL+ALT+Del to start the task manager. Then select the Processes tab. In the menu View/Select Columns..., tick the box PID (Process Identifier), which will add a column PID to the task manager. Then find the (PID) corresponding to the name Rterm.exe *32 (e.g., 5404). An easier option is to type Sys.getpid() in the upper R windows of Emacs.

Linux

Under Linux, you can get the PID of R directly by typing in Emacs:

```
M-! Shell command: pgrep R [RET]
```

Then type in Emacs the following instructions:

```
(gdb) attach 5404 [RET]
(gdb) signal 0 [RET]
```

Click on the panel (or *Buffer* in Emacs) called *R*, and execute the following instructions:

```
> setwd("path to combn.R file")
> source("combn.R")
> dyn.load(paste("combn",.Platform$dynlib.ext,sep=""))
```

Click on the bottom sub-window (*Buffer* *gud*).

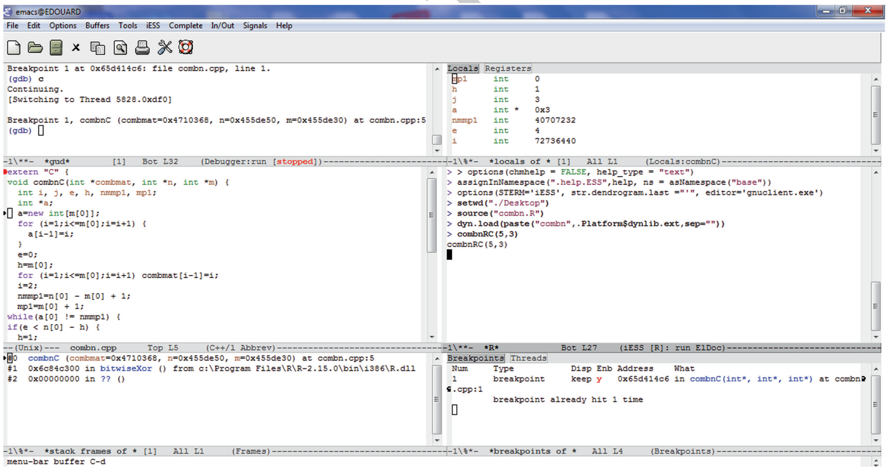
```
C-c C-c
(gdb) b combn.cpp:1 [RET]
(gdb) c [RET]
```

Click on the top sub-window (*Buffer* *R*).

```
> combnRC(5,3)
```

```
C-g
M-x gdb-many-windows
```

Put the Emacs window in full screen. Your Emacs window should now be divided in six panels, as shown in Fig. 8.2. If needed, click on the relevant entries of the Buffer menu.


emacs@EDOUARD
File Edit Options Buffers Tools iESS Complete In/Out Signals Help

```
Breakpoint 1 at 0x65d414c6: file combn.cpp, line 1.
(gdb) c
Continuing.
[Switching to Thread 5828.0xdf0]

Breakpoint 1, combnC (combmat=0x4710368, n=0x455de50, m=0x455de30) at combn.cpp:5
(gdb)
-1\**-  *gud*         [1]    Bot L32     (Debugger:run [stopped])--------------------
```

```
Locals Registers
 mp1      int      0
 h        int      1
 j        int      3
 a        int *    0x3
 nmmp1    int      40707232
 e        int      4
 i        int      72736440
-1\%*-  *locals of * [1]   All L1      (Locals:combnC)--------------------------
```

```
extern "C" {
void combnC(int *combmat, int *n, int *m) {
  int i, j, e, h, nmmp1, mp1;
  int *a;
  a=new int[m[0]];
  for (i=1;i<=m[0];i=i+1) {
    a[i-1]=i;
  }
  e=0;
  h=m[0];
  for (i=1;i<=m[0];i=i+1) combmat[i-1]=i;
  i=2;
  nmmp1=n[0] - m[0] + 1;
  mp1=m[0] + 1;
while(a[0] != nmmp1) {
if(e < n[0] - h) {
  h=1;
--(Unix)---  combn.cpp      Top L5      (C++/l Abbrev)----------------------------
```

```
> > options(chmhelp = FALSE, help_type = "text")
> assignInNamespace(".help.ESS",help, ns = asNamespace("base"))
> options(STERM='iESS', str.dendrogram.last ="'", editor='gnuclient.exe')
> setwd("./Desktop")
> source("combn.R")
> dyn.load(paste("combn",.Platform$dynlib.ext,sep=""))
> combnRC(5,3)
combnRC(5,3)

-1\**-  *R*           Bot L27     (iESS [R]: run ElDoc)-----------------------------
```

```
#0  combnC (combmat=0x4710368, n=0x455de50, m=0x455de30) at combn.cpp:5
#1  0x6c84c300 in bitwiseXor () from c:\Program Files\R\R-2.15.0\bin\i386\R.dll
#2  0x00000000 in ?? ()
-1\%*-  *stack frames of * [1]   All L1      (Frames)-------------------------------
menu-bar buffer C-d
```

```
Breakpoints Threads
 Num     Type           Disp Enb Address    What
 1       breakpoint     keep y   0x65d414c6 in combnC(int*, int*, int*) at combn
.cpp:1
         breakpoint already hit 1 time

-1\%*-  *breakpoints of *   All L4      (Breakpoints)-----------------------------
```

Fig. 8.2: Emacs and GDB

Click on the bottom right panel called *breakpoints of*. Select the menu Buffers/*R* *.

Now click on the window combn.cpp. You will see new icons in the top part of Emacs. For example, you can click on the symbol for Next Line (right of GO) to execute your C/C++ line by line.

Do it yourself

- Change line 32 of file integ.cpp into `limit[0] = -1;`. Recompile this code and call it from R as seen above: `.C("testintegral",val = 0.0)$val`. Your R session should crash. Suppose you do not remember making the above change. Use the techniques you just learnt to find the error.
- Debug the file pmvt.cpp seen in Sect. 8.5.4. Type the instruction `p Rf_PrintValue(Rpmvt)` from the GDB console to display (in the R console) the contents of object `Rpmvt`.

8.6.4.2 Debugging with DDD

You first need to launch `Xming` (or an equivalent tool); its icon should appear in the task bar. Then launch a `Cygwin` terminal window, and type the following instructions:

```
$ export DISPLAY=localhost:0.0
$ cd path to directory containing the source and DLL files
$ ddd Rgui
```

You may need to wait a while before DDD starts.

Linux

Under Linux, replace the last instruction with the command `R -d ddd`.

Next, type the following instructions in GDB (lower panel):

```
(gdb) dir $cwd
(gdb) run
```

The first instruction tells GDB to search for source files in the current directory (which would be given by the command `pwd`), thus avoiding issues due to path management in Windows. The second instruction starts R (you could also tick the box: `Program/Run in Execution Window`, and click on `Program/Run`, then on `Run`); type in R:

```
> dyn.load("inv.dll")
```

Note that the file inv.dll was created with debugging information, as mentioned page 259. Now go to menu `Misc/Break to debugger` to return to DDD. Go to menu `File/Open source...` and open file inv.cpp. Also tick the entry `Data Window` in menu `View` (and possibly entry `Display Local Variables` in menu `Data`, if you are patient!). You can then put one or several breaking points in the code to debug (by double-clicking at the beginning of the line or by right-clicking), for example, at the instruction `M << values;`. This has the effect of displaying a stop symbol. Then type `continue` (or just `c`) in the lower part `(gdb)`. This returns to `R`, where you type

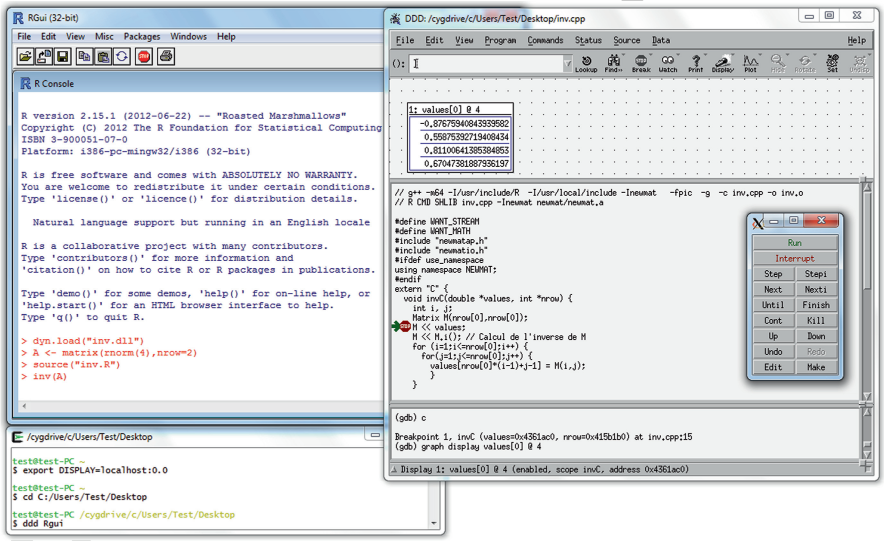
```
> A <- matrix(rnorm(4),nrow=2)
> source("inv.R") # File created page 247.
> inv(A)
```

When the (first) breaking point is encountered by the processor, code execution is suspended. You can now use the graphical tool `DDD` to debug your code.

Note that it is possible to display several values of an array. For example, you can type in the lower window `(gdb)` the following instruction (Fig. 8.3):

```
graph display values[0] @ 4
```

to display the (first) four values of array `values`.


RGui (32-bit)
File Edit View Misc Packages Windows Help
R Console
DDD: /cygdrive/c/Users/Test/Desktop/inv.cpp
File Edit View Program Commands Status Source Data Help
():
Lookup Find» Break Watch Print Display Plot Hide Rotate Set Undisp
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make
/cygdrive/c/Users/Test/Desktop

```
R version 2.15.1 (2012-06-22) -- "Roasted Marshmallows"
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: i386-pc-mingw32/i386 (32-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> dyn.load("inv.dll")
> A <- matrix(rnorm(4),nrow=2)
> source("inv.R")
> inv(A)
```

```
1: values[0] @ 4
-0.87675940843939582
 0.55875392719408434
 0.81100641385384853
 0.67047381887936197
```

```
// g++ -m64 -I/usr/include/R  -I/usr/local/include -Inewmat   -fpic  -g  -c inv.cpp -o inv.o
// R CMD SHLIB inv.cpp -Inewmat newmat/newmat.a

#define WANT_STREAM
#define WANT_MATH
#include "newmatap.h"
#include "newmatio.h"
#ifdef use_namespace
using namespace NEWMAT;
#endif
extern "C" {
  void invC(double *values, int *nrow) {
    int i, j;
    Matrix M(nrow[0],nrow[0]);
    M << values;
    M << M.i(); // Calcul de l'inverse de M
    for (i=1;i<=nrow[0];i++) {
      for(j=1;j<=nrow[0];j++) {
        values[nrow[0]*(i-1)+j-1] = M(i,j);
      }
    }
```

```
(gdb) c

Breakpoint 1, invC (values=0x4361ac0, nrow=0x415b1b0) at inv.cpp:15
(gdb) graph display values[0] @ 4
```

```
Display 1: values[0] @ 4 (enabled, scope invC, address 0x4361ac0)
```

```
test@test-PC ~
$ export DISPLAY=localhost:0.0

test@test-PC ~
$ cd C:/Users/Test/Desktop

test@test-PC /cygdrive/c/Users/Test/Desktop
$ ddd Rgui
```

Fig. 8.3: DDD and GDB

8.6.4.3 Debugging with `Insight`

`Insight` seems to have difficulties working on some Windows versions. Nonetheless, we present this software for those who have a compatible version of Windows, or in case a new version of `Insight` is shipped after the publication of this book.

Recompile your file using flag `-g` (and possibly `-fPIC`) which tells the C++ compiler to add information on the source code directly in the compiled file.

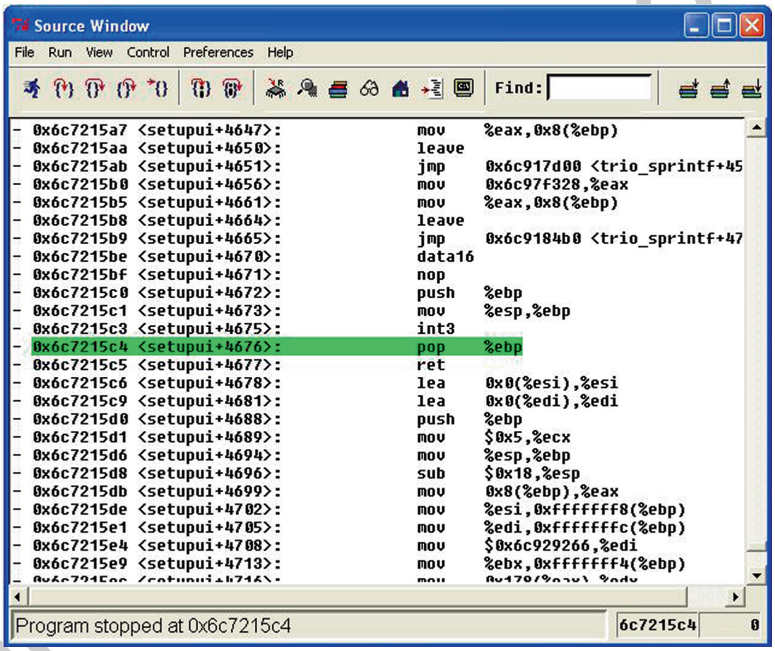
```
g++ -c combn.cpp -o combn.o -g
g++ -shared -o combn.dll combn.o
```

Then, from the `MS-DOS` window, execute `insight Rgui.exe`, then click on `Run` .

Next type the following commands in the R console which opens:

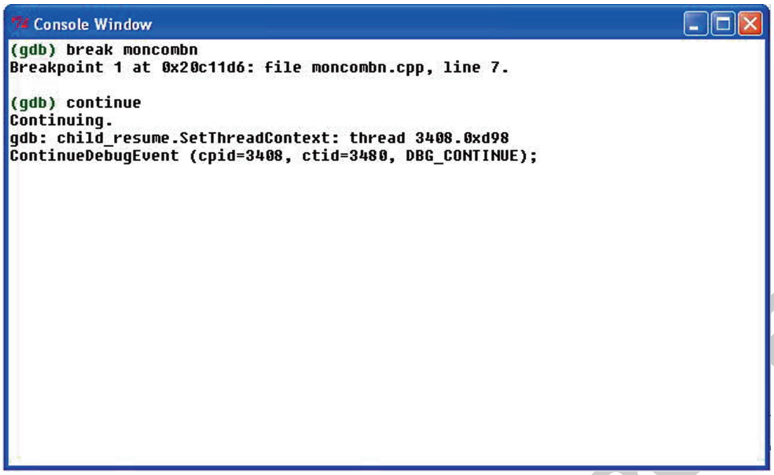
```
> source("combn.R")
> dyn.load(paste("combn",.Platform$dynlib.ext,sep=""))
```

Go to the R menu called `Misc`, then `Break to debugger`. You are now in the `Insight` window.



```
- 0x6c7215a7 <setupui+4647>:      mov    %eax,0x8(%ebp)
- 0x6c7215aa <setupui+4650>:      leave
- 0x6c7215ab <setupui+4651>:      jmp    0x6c917d00 <trio_sprintf+45
- 0x6c7215b0 <setupui+4656>:      mov    0x6c97f328,%eax
- 0x6c7215b5 <setupui+4661>:      mov    %eax,0x8(%ebp)
- 0x6c7215b8 <setupui+4664>:      leave
- 0x6c7215b9 <setupui+4665>:      jmp    0x6c9184b0 <trio_sprintf+47
- 0x6c7215be <setupui+4670>:      data16
- 0x6c7215bf <setupui+4671>:      nop
- 0x6c7215c0 <setupui+4672>:      push   %ebp
- 0x6c7215c1 <setupui+4673>:      mov    %esp,%ebp
- 0x6c7215c3 <setupui+4675>:      int3
- 0x6c7215c4 <setupui+4676>:      pop    %ebp
- 0x6c7215c5 <setupui+4677>:      ret
- 0x6c7215c6 <setupui+4678>:      lea    0x0(%esi),%esi
- 0x6c7215c9 <setupui+4681>:      lea    0x0(%edi),%edi
- 0x6c7215d0 <setupui+4688>:      push   %ebp
- 0x6c7215d1 <setupui+4689>:      mov    $0x5,%ecx
- 0x6c7215d6 <setupui+4694>:      mov    %esp,%ebp
- 0x6c7215d8 <setupui+4696>:      sub    $0x18,%esp
- 0x6c7215db <setupui+4699>:      mov    0x8(%ebp),%eax
- 0x6c7215de <setupui+4702>:      mov    %esi,0xfffffff8(%ebp)
- 0x6c7215e1 <setupui+4705>:      mov    %edi,0xfffffffc(%ebp)
- 0x6c7215e4 <setupui+4708>:      mov    $0x6c929266,%edi
- 0x6c7215e9 <setupui+4713>:      mov    %ebx,0xfffffff4(%ebp)

Program stopped at 0x6c7215c4                 6c7215c4     0
```

In `Insight`, select menu `View - Console [CTRL+N]`. This opens the command window of debugger GDB. We can now add a breaking point to function `combnC` by typing
`break combnC`
Then type:
`continue`
which returns to R. As soon as the function `combnC` is called, we will return to the debugger.

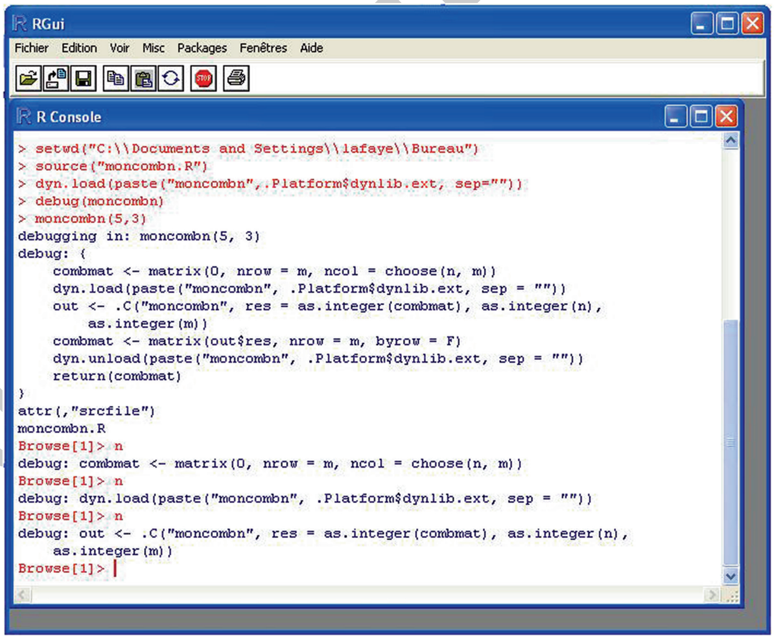


```
(gdb) break moncombn
Breakpoint 1 at 0x20c11d6: file moncombn.cpp, line 7.

(gdb) continue
Continuing.
gdb: child_resume.SetThreadContext: thread 3408.0xd98
ContinueDebugEvent (cpid=3408, ctid=3480, DBG_CONTINUE);
```

Now type in R:

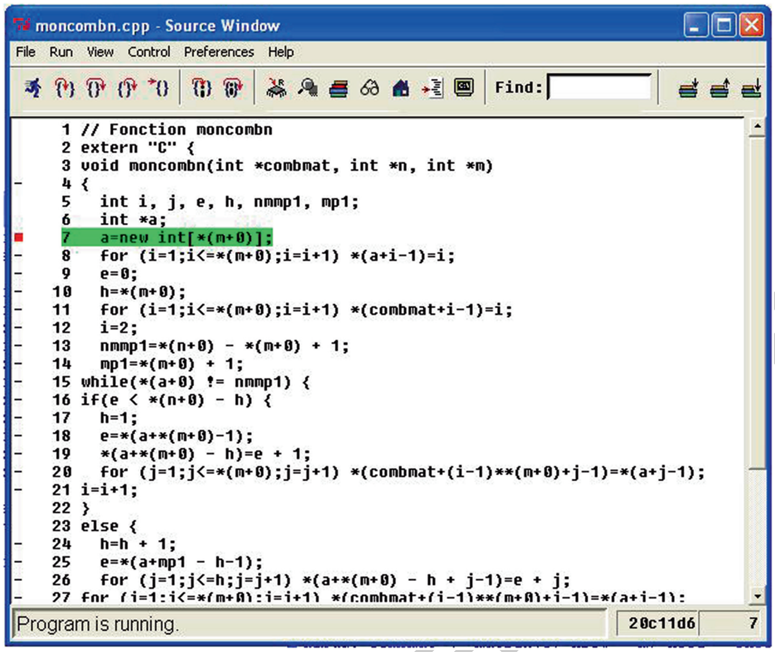
```
> debug(combnRC)
> combnRC(5,3)
```

Use instruction n (for *next*) to skip to the next instruction of our R code, until reaching the call to the function written in C++.



```
> setwd("C:\\Documents and Settings\\lafaye\\Bureau")
> source("moncombn.R")
> dyn.load(paste("moncombn",.Platform$dynlib.ext, sep=""))
> debug(moncombn)
> moncombn(5,3)
debugging in: moncombn(5, 3)
debug: {
    combmat <- matrix(0, nrow = m, ncol = choose(n, m))
    dyn.load(paste("moncombn", .Platform$dynlib.ext, sep = ""))
    out <- .C("moncombn", res = as.integer(combmat), as.integer(n),
        as.integer(m))
    combmat <- matrix(out$res, nrow = m, byrow = F)
    dyn.unload(paste("moncombn", .Platform$dynlib.ext, sep = ""))
    return(combmat)
}
attr(,"srcfile")
moncombn.R
Browse[1]> n
debug: combmat <- matrix(0, nrow = m, ncol = choose(n, m))
Browse[1]> n
debug: dyn.load(paste("moncombn", .Platform$dynlib.ext, sep = ""))
Browse[1]> n
debug: out <- .C("moncombn", res = as.integer(combmat), as.integer(n),
    as.integer(m))
Browse[1]> 
```

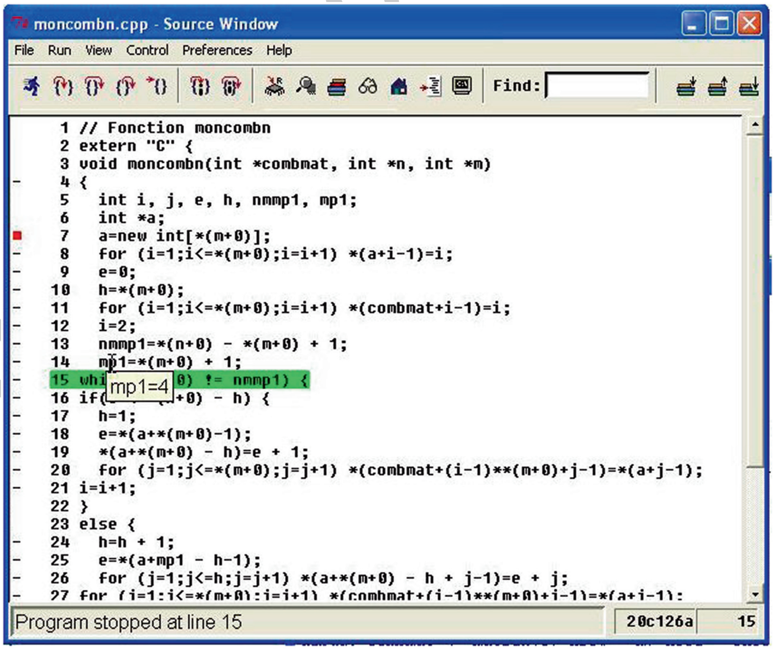
The breaking point we added is detected and we are back in Insight.


moncombn.cpp - Source Window
File Run View Control Preferences Help
Find:

```
 1 // Fonction moncombn
 2 extern "C" {
 3 void moncombn(int *combmat, int *n, int *m)
 4 {
 5   int i, j, e, h, nmmp1, mp1;
 6   int *a;
 7   a=new int[*(m+0)];
 8   for (i=1;i<=*(m+0);i=i+1) *(a+i-1)=i;
 9   e=0;
10   h=*(m+0);
11   for (i=1;i<=*(m+0);i=i+1) *(combmat+i-1)=i;
12   i=2;
13   nmmp1=*(n+0) - *(m+0) + 1;
14   mp1=*(m+0) + 1;
15 while(*(a+0) != nmmp1) {
16 if(e < *(n+0) - h) {
17   h=1;
18   e=*(a+*(m+0)-1);
19   *(a+*(m+0) - h)=e + 1;
20   for (j=1;j<=*(m+0);j=j+1) *(combmat+(i-1)**(m+0)+j-1)=*(a+j-1);
21 i=i+1;
22 }
23 else {
24   h=h + 1;
25   e=*(a+mp1 - h-1);
26   for (j=1;j<=h;j=j+1) *(a+*(m+0) - h + j-1)=e + j;
27 for (i=1;i<=*(m+0);i=i+1) *(combmat+(i-1)**(m+0)+i-1)=*(a+i-1);
```

```
Program is running.          20c11d6    7
```

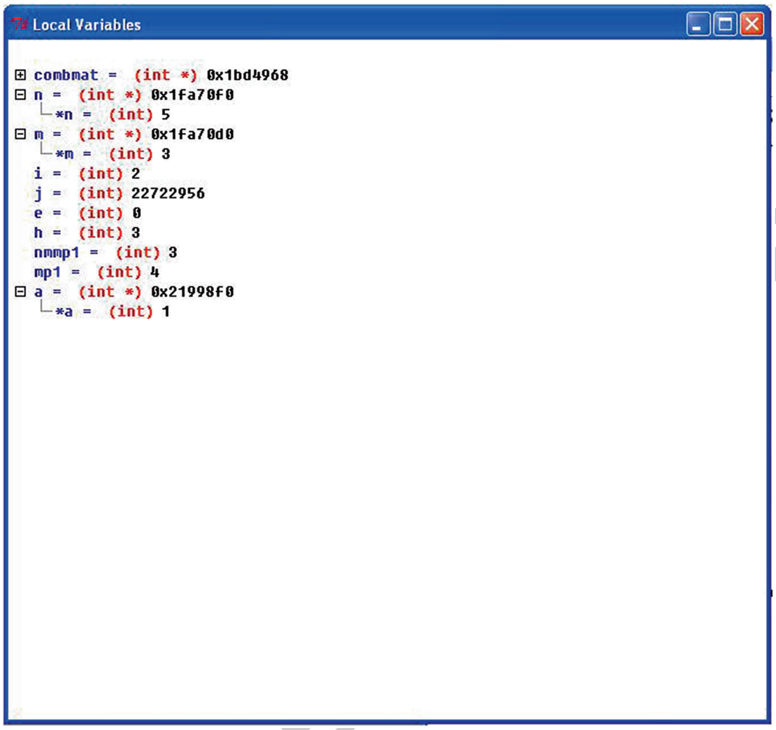
Next click on icon {}↓ to execute line by line the C++ code and check the value of the various variables.


moncombn.cpp - Source Window
File Run View Control Preferences Help
Find:
mp1=4

```
 1 // Fonction moncombn
 2 extern "C" {
 3 void moncombn(int *combmat, int *n, int *m)
 4 {
 5   int i, j, e, h, nmmp1, mp1;
 6   int *a;
 7   a=new int[*(m+0)];
 8   for (i=1;i<=*(m+0);i=i+1) *(a+i-1)=i;
 9   e=0;
10   h=*(m+0);
11   for (i=1;i<=*(m+0);i=i+1) *(combmat+i-1)=i;
12   i=2;
13   nmmp1=*(n+0) - *(m+0) + 1;
14   mp1=*(m+0) + 1;
15 whi         0) != nmmp1) {
16 if(        +0) - h) {
17   h=1;
18   e=*(a+*(m+0)-1);
19   *(a+*(m+0) - h)=e + 1;
20   for (j=1;j<=*(m+0);j=j+1) *(combmat+(i-1)**(m+0)+j-1)=*(a+j-1);
21 i=i+1;
22 }
23 else {
24   h=h + 1;
25   e=*(a+mp1 - h-1);
26   for (j=1;j<=h;j=j+1) *(a+*(m+0) - h + j-1)=e + j;
27 for (i=1;i<=*(m+0);i=i+1) *(combmat+(i-1)**(m+0)+i-1)=*(a+i-1);
```

```
Program stopped at line 15          20c126a    15
```

The window `Local Variables` (shown by menu `View -> Local Variable` 1401
`[CTRL+L]`) displays all local variables and their contents during code execution. 1402



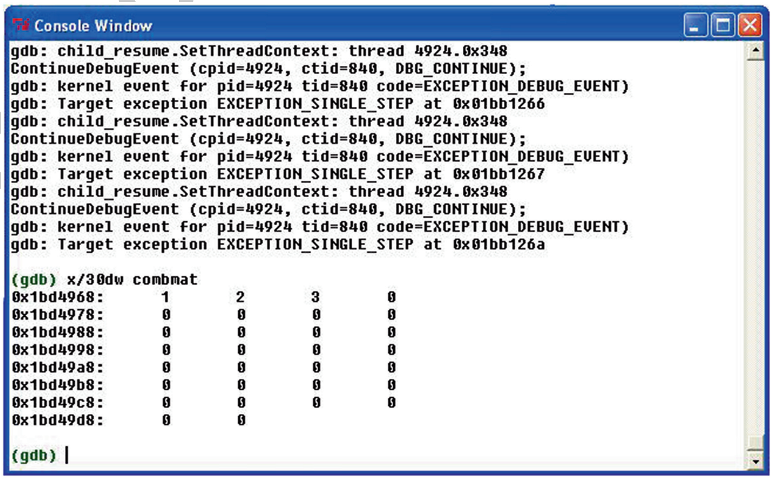
Local Variables

```
⊞ combmat =  (int *) 0x1bd4968
⊟ n =  (int *) 0x1fa70f0
   └*n =  (int) 5
⊟ m =  (int *) 0x1fa70d0
   └*m =  (int) 3
  i =  (int) 2
  j =  (int) 22722956
  e =  (int) 0
  h =  (int) 3
  nmmp1 =  (int) 3
  mp1 =  (int) 4
⊟ a =  (int *) 0x21998f0
   └*a =  (int) 1
```

1403

Note that to see the contents of an R matrix or vector, you simply need to go to the 1404
GDB console and type for example: 1405

```
x/30dw combmat
```
1406



Console Window

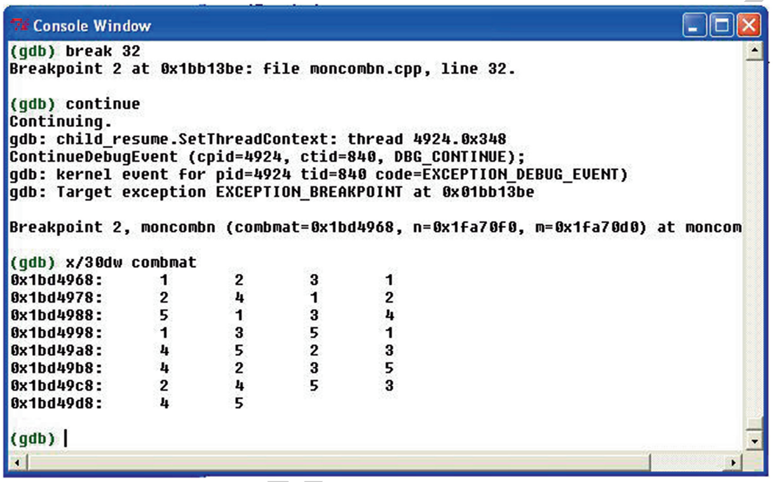
```
gdb: child_resume.SetThreadContext: thread 4924.0x348
ContinueDebugEvent (cpid=4924, ctid=840, DBG_CONTINUE);
gdb: kernel event for pid=4924 tid=840 code=EXCEPTION_DEBUG_EVENT)
gdb: Target exception EXCEPTION_SINGLE_STEP at 0x01bb1266
gdb: child_resume.SetThreadContext: thread 4924.0x348
ContinueDebugEvent (cpid=4924, ctid=840, DBG_CONTINUE);
gdb: kernel event for pid=4924 tid=840 code=EXCEPTION_DEBUG_EVENT)
gdb: Target exception EXCEPTION_SINGLE_STEP at 0x01bb1267
gdb: child_resume.SetThreadContext: thread 4924.0x348
ContinueDebugEvent (cpid=4924, ctid=840, DBG_CONTINUE);
gdb: kernel event for pid=4924 tid=840 code=EXCEPTION_DEBUG_EVENT)
gdb: Target exception EXCEPTION_SINGLE_STEP at 0x01bb126a

(gdb) x/30dw combmat
0x1bd4968:      1       2       3       0
0x1bd4978:      0       0       0       0
0x1bd4988:      0       0       0       0
0x1bd4998:      0       0       0       0
0x1bd49a8:      0       0       0       0
0x1bd49b8:      0       0       0       0
0x1bd49c8:      0       0       0       0
0x1bd49d8:      0       0

(gdb) |
```

1407

You can also display graphically this table of values and select it by clicking on `plot`.

You can now type the following instructions in the GDB console to add a breaking point at line 32 of your C++ code, then reexecute the code. When the breaking point is encountered, the code stops again and we can ask to display again the contents of array x:

```
break 32
continue
x/30dw combmat
```



```
(gdb) break 32
Breakpoint 2 at 0x1bb13be: file moncombn.cpp, line 32.

(gdb) continue
Continuing.
gdb: child_resume.SetThreadContext: thread 4924.0x348
ContinueDebugEvent (cpid=4924, ctid=840, DBG_CONTINUE);
gdb: kernel event for pid=4924 tid=840 code=EXCEPTION_DEBUG_EVENT)
gdb: Target exception EXCEPTION_BREAKPOINT at 0x01bb13be

Breakpoint 2, moncombn (combmat=0x1bd4968, n=0x1fa70f0, m=0x1fa70d0) at moncom

(gdb) x/30dw combmat
0x1bd4968:      1       2       3       1
0x1bd4978:      2       4       1       2
0x1bd4988:      5       1       3       4
0x1bd4998:      1       3       5       1
0x1bd49a8:      4       5       2       3
0x1bd49b8:      4       2       3       5
0x1bd49c8:      2       4       5       3
0x1bd49d8:      4       5

(gdb) |
```

8.6.4.4 Detecting Memory Leaks

The messages `Segmentation fault` (or `segfault`), `invalid next size`, `std::bad_alloc` (which you will certainly encounter under Linux!), incoherent results or, more radically, a complete crash of R are often indications that there is a memory issue (access to a non-reserved or non-initialized address, using freed memory, etc.) These memory leaks often occur when you have forgotten to use the instruction `delete[] ptr;` to delete from memory a pointer `ptr` introduced in a C/C++ function. This problem can sometimes be noticed in the task manager when you run a large simulation in R and realize that the R process is using more and more memory even though it should not.

Linux

Under Linux, the display of memory usage by different processes is given by the command (entered in a terminal window) `watch -d free` for global usage or by `top -p PID` for a specific process (use `ps au` to find the PID of the desired process). You can also use the graphical tool `ksysguard`.

Another common mistake is to try to manipulate the *n*th entry in an array of size less than *n* (accessing undefined memory). It can then be difficult to detect the origin of the problem. The software `Dr Memory` (`http://code.google.com/p/drmemory`) and possibly the software `electric-fence-win32` (`http://code.google.com/p/electric-fence-win32`) and `duma` (`http://duma.sourceforge.net`) can be precious tools in such situations.

Linux

Under Linux, you can use the software `Valgrind` or `Electric Fence`.

We now show on an example how to use `Dr. Memory` which you should install in the directory `C:\drmemory` (choose the entry `Add Dr. Memory to the system PATH for all users` upon installation).

The following piece of code includes several errors, which can be hard to find for a beginner. You can download it from `http://biostatisticien.eu/springeR/memory.cpp`.

```
extern "C" {
  void testmemory(int *M, double *a) {
    double *ptr1, *ptr2;
    int i;
    ptr1 = new double[10000];
    ptr2 = new double[M[0]];
    ptr1[0] = 1.0;
    for (i=1;i<10000;i++) {
      ptr1[i] = (double)i;
      ptr2[i] = ptr1[i-1] * (double)i;
    }
    delete[] ptr2;
    for (i=0;i<10;i++) a[i] = ptr2[i];
    return;
  }
}
```

First create the associated DLL file, using the following instructions in an `Ms-Dos` window:

```
cd directory containing file memory.cpp
g++ -o memory.o -c memory.cpp -g
g++ -shared -o memory.dll memory.o
```

Linux

Under Linux, use the instructions:

```
g++ -o memory.o -c memory.cpp -g -fPIC
g++ -shared -o memory.so memory.o
```

Next, type `drmemory.exe -- Rgui` in your command window (be patient), then type the following instructions in the R console:

```
> dyn.load("memory.dll")
> .C("testmemory",10000L,3.0)
> q()
```

Now look for the instances of `testmemory` in the file which opened up. This will indicate the lines which may contain errors. For example, this shows that there is an error at line 13. In fact, we realize that the array `a` is of length 1 (and initially contains only the value 3.0), whereas we are trying to write values in entries 0–9. Furthermore, the pointer `ptr2` was deleted on the preceding line.

You can also try the following R instruction, and note in the task manager that the amount of RAM used by R increases greatly. This is because we forgot the instruction `delete[] ptr1;` in the C/C++ code above:

```
> for (i in 1:10000) .C("testmemory",10000L,as.double(1:10))
```

Linux

The equivalent of `Dr Memory` under Linux is called `Valgrind`. To detect where the leak comes from, you can use the instruction:

```
R -d 'valgrind --leak-check=full'
```

```
> dyn.load("memory.so")
> .C("testmemory",10000L,3.0)
> q()
```

In the output of `valgrind`, you then need to look for the errors and for the corresponding line numbers in the source code of `memory.cpp`. The following instructions give other error types displayed by R and detected by `Valgrind`:

```
> # Works only once!
> # Afterwards, R crashes with: "caught segfault":
> .C("testmemory",10000L,c(3.0,5.0))
> # R closes: "invalid next size":
> .C("testmemory",10000,c(3.0,5.0))
> # R closes: "std::bad_alloc":
> .C("testmemory",10^12,c(3.0,5.0))
> # Works when ptr2 is no longer defined:
> .C("testmemory",10000L,as.double(1:10))
```

SECTION 8.7

Parallel Computing and Computation on Graphical Cards

8.7.1 Parallel Computing

You can speed up your calculations by having them run on several processors at the same time; these processors can even be on different computers. There are several specialized packages for this; they are listed in the `CRAN Task View: High-Performance and Parallel Computing with R`, available at `http://cran.r-project.org/web/views/HighPerformanceComputing.html`.

The easiest to use is package `parallel` with communication protocol `PSOCK`, which we briefly describe below through an example.

Tip

The `MPI` protocol (*Message Passing Interface*), used by package `Rmpi`, is more flexible than the `PSOCK` protocol, but it requires the installation of other software (such as `OpenMPI` or `mpich2`).

See also

We refer the interested reader to the websites `http://www.divms.uiowa.edu/~luke/R/cluster/cluster.html`, `http://www.sfu.ca/~sblay/R/snow.html` and `http://cran.r-project.org/web/packages/snowfall/vignettes/snowfall.pdf`.

The following `R` code performs numerical evaluation (by Monte Carlo simulation) of the empirical level of the Shapiro-Wilks normality test for a nominal level of 5 %:

```
> myfunc <- function(M=1000) {
+  decision <- 0
+  for (i in 1:M) {
+   x <- rnorm(100)
+   if (shapiro.test(x)$p < 0.05) decision <- decision + 1
+  }
+ return(decision)
+ }
```

Here is the computation time needed for this code with $M = 60,000$ Monte Carlo iterations:

```
> system.time({
+  M <- 60000
+   decision <- myfunc(M)
+   print(decision/M)
+ })
[1] 0.04893333
   user  system elapsed
 18.124   0.331  18.457
```

We now show how this code can be parallelized using the package parallel 1490
and the corresponding gain in computation time. We used six processors. 1491

Tip

To know the number of processors on your computer, type the instruction devmgmt.msc in the menu Start/Run. Then count the number of lines in the Processors entry. Under Linux, type top in a terminal window, then 1. This shows the number of processors. Another option is to use the function detectCores() of package parallel.

```
> require("parallel")
> system.time({
+   nbclus <- 6
+   M <- 60000
+   cl <- makeCluster(nbclus, type = "PSOCK")
+   out <- clusterCall(cl, myfunc, round(M/nbclus))
+   stopCluster(cl)
+   decision <- 0
+   for (clus in 1:nbclus) {
+    decision <- decision + out[[clus]]
+   }
+   print(decision/(round(M/nbclus)*nbclus))
+ })
[1] 0.0501
   user  system elapsed
  0.019   0.033   5.522
```

8.7.2 Computation on Graphical Cards 1492

The processor, or CPU (*central processing unit*), is the computer component which 1493
handles execution of software. However, it is now also possible to perform com- 1494
putations on a GPU (*graphical processing unit*), or graphical card. Graphical cards 1495
allowing such operations are marketed by Nvidia, and they can include hundreds of 1496
processors working in parallel. The speed-up in computation time can be substan- 1497
tial. To use this technology, however, you must know the programming language 1498
CUDA, developed by Nvidia. A few R developers have delved into this language 1499
and have grouped a few functions in the package gputools, which is only available 1500
on Linux for now. 1501

Here is a short example of use of this package. We used an NVIDIA GeForce 1502
GTX 480 graphical card. 1503

```
> require("gputools")
> A <- matrix(runif(40000),nrow=200,ncol=200)
> B <- matrix(runif(40000),nrow=200,ncol=200)
> system.time(cor(A, B, method="kendall")) # Computation CPU.
   user  system elapsed
 29.804   0.002  29.810
> system.time(gpuCor(A, B, method="kendall")) # Computation on
                                               # GPU.
   user  system elapsed
  0.836   0.052   0.891
```

See also

To find out more on this topic, go to http://cran.r-project.org/web/packages/gputools/gputools.pdf and http://developer.nvidia.com/object/cuda_training.html.

Memorandum

`function(<par1>,<par2>,...,<parN>) <body>` : declare a function object
`"{"()`: define a block of instructions and return the last evaluated instruction
`class(), "class<-"()`: extract, affect the class of an object
`missing()`: test whether an effective argument is defined
`attributes(), "attributes<-"()`: extract, affect all attributes as a list
`attr(), "attr<-"()`: extract, affect a single attribute
`expression()`: create an expression object
`parse()`: convert text to an expression
`eval()`: evaluate an expression
`"∼"()`: create a formula object
`new.env()`: create an environment
`local()`: execute code locally in an environment

Exercises

8.1- For each of the following command lines, indicate the class of the returned R object. What is displayed upon execution of each of these command lines?

- `function(name) {name}`
- `(function(name) {name})("Ben")`
- `(function(name) {cat(name,"\n")})("Ben")`
- `(function(name) {invisible(name)})("Ben")`

8.2- Is there a difference between

- `name <- function(name) name` and `name <- function(name) {name}`
- `name <- function(name) {name}` and `name <- function(name) {return(name)}`
- `name <- function(name) {name}` and `(function(name) {name}) -> name`

8.3- Upon execution, is there a difference between `name()` and `name("Peter")` when

- `name <- function(name="Peter") name`
- `name <- function(name="Peter") name2 <- name`

For these two declarations of the function `name()`, is there a difference in the type of the R object `res` given by `res <- name("Ben")`?

8.4- What R object is returned upon execution of `name()` when

```
name <- function(name="Peter") {
  name
  # The last instruction is a comment!
}
```

8.5- When `name <- function(firstname="Peter",name="L") { paste(firstname,name)}`, what R object is returned by

- `name(firstname="Ben")`
- `name(fir="Ben")`
- `name(n="D",f="R")`

8.6- Rewrite the following function declaration in one line, without using the command separator “`;`”:
`name <- function(name) { if(missing("name"))`
`name <- "Peter"; cat(name,"\n") }`

8.7- What is the output of the execution of `nameS("peteR","Ben","R")` when

- `nameS <- function(...) c(...)`
- `nameS <- function(...) list(...)`
- `nameS <- function(...) for(name in c(...)) print(name)`
- `nameS <- function(...) for(name in list(...))`
  `print(name)`

Same question upon execution of
`nameS(c("peteR","L"),c("Ben","L"),c("R","D"))`

8.8- When `nameS <- function(names=c("Ben","R"),...) c(names,...)`, which R objects are returned by `nameS("PeteR")`, `nameS(name="PeteR")` and `nameS(names="PeteR")`? Same question when
`nameS <- function(...,names=c("Ben","R")) c(names,...)`.

8.9- Create a constructor function `Male()` generating an object of class `"Male"` with fields `firstname` and `name` (in an object of type `list`). Create the method `hello.Male()` which displays `"Hello Mister FIRSTNAME NAME!"` (do not forget the "$\backslash n$" at the end of the display!) for an object with values `"FIRSTNAME"` and `"NAME"`, respectively, for the fields `firstname` and `name`. When `man <- Male("Ben","L")`, what is produced upon execution of the following commands: `hello.Male(man)` and `hello(man)`? What code should you execute in addition for the two results to be identical?

8.10- Create the analogous functions for the class `"Female"` (hint: do not forget to update the gender in `hello.Female()`). When
`woman <- Female("Elsa","R")`, what is produced upon execution of the following commands: `hello.Male(woman)`, `hello.Female(woman)` and `hello(woman)`?

8.11- When `welcome <- function(...) for(person in list(...)){ hello(person)}`, what is returned by `welcome(man,woman)`?
And when `welcome <- function(...) for(person in c(...)){ hello(person)}`?
Same question when `hello.default <- function(obj){ cat("hello",obj,"!\n")}`.

Worksheet

Programming Functions and Object-Oriented Programming in R

Before reading the practicals of this chapter, we strongly advise you to revise those of the previous chapters (especially the one on "advanced plots") and to reorganize their solutions in as many functions as necessary.

A- Managing a Bank Account

The aim of this practical is to create three minimalist functions to manage bank accounts. The accounts will be stored in data.frame objects all called `accounts` and stored in different .RData files. All these files will be located in the same folder. The path to this folder should be saved in the R variable `.folder.accounts` and be accessible in all the functions you develop.

8.1- The instruction `file.path(.folder.accounts,paste(name,".RData", sep=""))` gives the path of the file associated with the account `Name`. Create the functions `path.account()`, which takes one formal argument `name` (representing the name of the account) and returns the complete path to the file (which contains the object `account` of class `data.frame`) with extension `.RData`.

8.2- Given that `factor(levels=c("Debit","Credit"))`, `numeric(0)` and `character(0)`, respectively, give empty vectors of explicit types, which expression would generate an empty data matrix with the predefined fields `amount`, `mode`, `date` and `remark`? Create the function `account()` (not to be confused with the variable `account` called in its body) which takes one argument `name` and creates a new account.

8.3- Create the functions `debit()` and `credit()` to, respectively, debit and credit an amount `amount` (second argument) from the account `name` (first argument). The third argument is any comment to put as `remark`. A fourth argument can represent the date; the default value is
`format(Sys.time(),"\%d/\%m/\%Y")` (*i.e.* the date of input). Remember to use the functions `load()` and `save()` to load and save the variable `account` in the body of each function.

8.4- If `account` is the data matrix containing information on the account, what is returned by `sum(account[account$mode=="Credit","amount"])`? Modify the function `account()` so that it returns the current state of the account only when the file returned by `path.account(name)` exists (use the function `file.exists()` to test whether a file exists).

8.5- Complete account management by creating any additional functions you wish.

8.6- Optional question: Since most use of R is done with objects, adapt the previous functions in a way that respects the R object-oriented philosophy. You can use the next practicals for inspiration.

B- Organizing Graphical Objects

When you think about it, plots in R do not really respect the object-oriented spirit: unlike most other entities, an R plot is not considered as an object which can be saved (and possibly modified) and on which certain methods can be applied. We shall attempt to propose a very basic prototype to draw a plot with circles and rectangles (and hence squares). You can enrich this library with graphical objects as you wish. Our aim is to maintain a list of graphical objects, with the possibility of changing any of its elements at any time.

8.1- R functions `plot.new()` and `plot.window()` are used to initialize a plot. The argument `asp` set to 1 creates plots with correct units for the *x* and *y* axes. Propose an object `Window` which gives the user the option of saving the dimensions of the graphics display window. The user can then call the constructor function (or method) `Window()` (which could have the same name as the class), which takes as arguments `x` and `y` (the coordinates of the centre), `width`, `height` (dimensions along the *x* and *y* axes, respectively) and optionally `log` (logarithmic transformation). All these quantities should be stored in an object `list`, returned by the constructor function `Window()`, after affecting its class to `"Window"`.

8.2- Similarly, propose constructor functions for objects of classes `Circle` and `Rectangle`. The fields `x` and `y` represent the coordinates of the centre of the object, `radius` is the radius of a circle and `width` and `height` are the dimensions of a rectangle.

8.3- Propose plotting methods `plot.Window()`, `plot.Rectangle()` and `plot.Circle()`. You can find inspiration in the following R treatments used to display a new plot with a circle and a square centred at the origin and of diameter and side length set to 1:

```
plot.new()
plot.window(xlim=c(-1,1),ylim=c(-1,1),asp=1)
rect(-.5,-.5,.5,.5)
symbols(0,0,circle=.5,inches=FALSE,add=TRUE)
```

8.4- Test the code you have developed by executing the code:

```
mywindow <- Window(0,0,2,2)
mycircle <- Circle(0,0,.5)
myrectangle <- Rectangle(0,0,1,1)
plot(mywindow);plot(mycircle);plot(myrectangle)
```

If all goes well, you should see a graphics window with a circle inside a square.

8.5- We now need to develop the methods associated with the class `MyPlot` which will contain the list of all graphical objects. First, propose a constructor function `MyPlot()` which initializes an object as `list(objects=list())` (where `objects` is the field containing the list of graphical objects), affects the class `"MyPlot"` and returns the object.

8.6- Propose a method add.MyPlot() which adds graphical objects. Remember to give a generic function add() to launch all associated methods. Use the functionalities of the list of supplementary arguments ... and the function c() so that the method add.MyPlot() can initialize as many graphical objects as the user wishes. Propose a method plot.MyPlot() which executes the methods plot() for all graphical objects. The user can then enter the following lines to get the same result as earlier:

```
myplot <- MyPlot()
myplot <- add(myplot,Window(0,0,2,2),Circle(0,0,.5),
                  Rectangle(0,0,1,1))
plot(myplot)
```

8.7- To display a plot, you need to initialize an object of type Window and put it in first position of the list of graphical objects of the class MyPlot. It might be useful to initialize it directly inside the constructor function MyPlot(). The arguments of the function Window() can be proposed directly for the function MyPlot(). Another idea is to propose a list of graphical objects to the user upon creation of an object of class MyPlot. As we have done for the method add.MyPlot(), we could use the list of supplementary arguments ..., which must be placed as first argument of the function MyPlot() so as to get the previous result with only two lines:

```
myplot <- MyPlot(Circle(),Rectangle())
plot(myplot)
```

However, note that in the first line, it is assumed that the default values of the arguments of the function Window(), Circle() and Rectangle() are appropriate.

8.8- The project is launched with this first prototype. You can complete it as you wish. If you need inspiration, you could try managing the list of graphical objects (e.g., deleting or modifying an object), display styles, axes, etc.

C- Creating a Class lm2 for Linear Regression with Two Regressors

The aim of this practical is to reproduce the procedure used by our two friends for simple regression. Graphical display will be made possible by the excellent package rgl, which is an OpenGL interface for R. Given the technical difficulty of this chapter, we propose here to develop functions (actually methods). Given that some aspects are very technical, the aim is only to get the reader to understand all the development steps of the following functions. This practical is aimed at more advanced users.

The following function returns an object of class lm2 which inherits from the standard class lm.

```
lm2 <- function(...) {
  obj <- lm(...)
  if(ncol(model.frame(obj))!=3)
    stop("two independent variables are required!")
  class(obj) <- c("lm2",class(obj)) # or c("lm2","lm")
  obj
}
```

For example, execute the following commands:

```
> n <- 20
> x1 <- runif(n,-5,5)
> x2 <- runif(n,-50,50)
> y <- 0.3+2*x1+2*x2+rnorm(n,0,20)
> reg2 <- lm2(y~x1+x2)
> summary(reg2)
Call:
lm(formula = ..1)
Residuals:
     Min       1Q   Median      3Q     Max
-32.0767 -17.1529   0.9872  12.3298  35.5909
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -1.8708     5.0769  -0.368    0.717
x1            2.8400     1.9594   1.449    0.165
x2            1.8084     0.1952   9.263  4.7e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 21.14 on 17 degrees of freedom
Multiple R-squared: 0.848,     Adjusted R-squared: 0.8301
F-statistic: 47.42 on 2 and 17 DF,  p-value: 1.112e-07
```

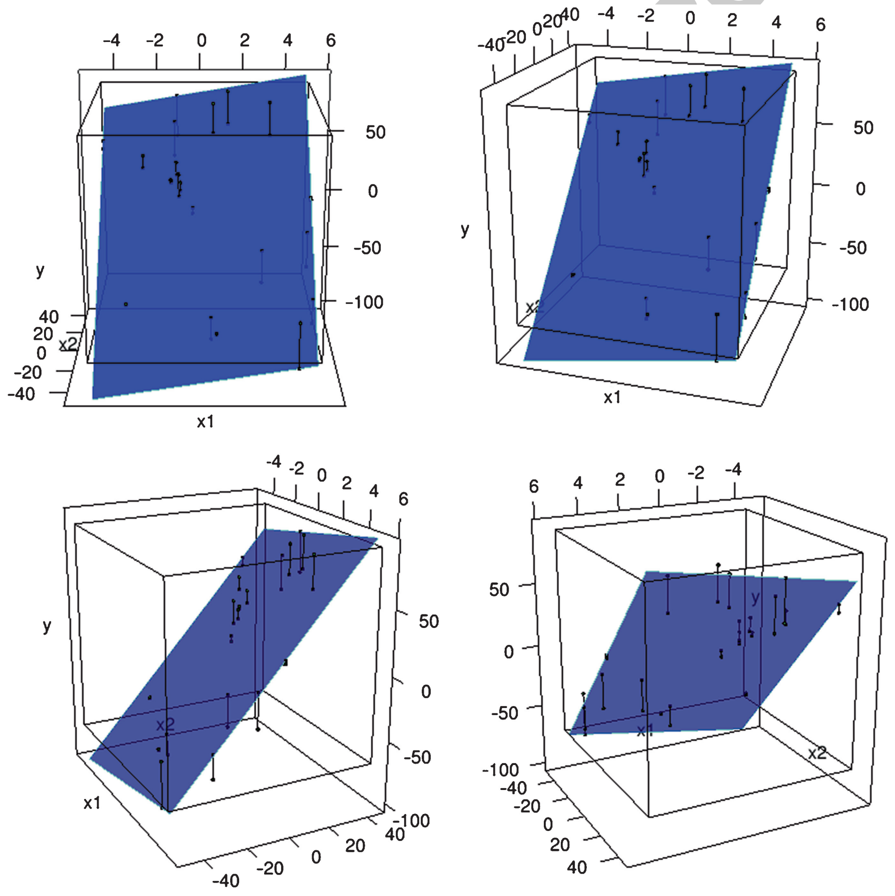
No surprises here: the R output of the summary is given by the method summary.lm().

The user now wishes a 3D scatter plot with the regression plane given by the standard method of least squares.

```
plot3d.lm2 <- function(obj,radius=1,lines=TRUE,
                       windowRect,...) {
  matreg <- model.frame(obj)
  colnames(matreg) <- c("y","x1","x2")
  predlim <- cbind(c(range(matreg[,2]),
                   rev(range(matreg[,2]))),
  rep(range(matreg[,3]),c(2,2)))
  predlim <- cbind(predlim,apply(predlim,1,
    function(l) sum(c(1,l)*coef(obj))
  ))
  if(missing(windowRect)) windowRect=c(2,2,500,500)
  open3d(windowRect=windowRect,...)
  bg3d(color = "white")
  plot3d(formula(obj),type="n")
  spheres3d(formula(obj),radius=radius,specular="green")
  quads3d(predlim,color="blue",alpha=0.7,shininess=128)
  quads3d(predlim,color="cyan",size=5,front="lines",
                back="lines",lit=F)
  if(lines) {
    matpred <- cbind(matreg[2:3],
                 model.matrix(obj)%*%coef(obj))
    points3d(matpred)
    colnames(matpred) <- c("x1","x2","y")
    matlines <- rbind(matreg[,c(2:3,1)],matpred)
    nr <- nrow(matreg)
    matlines <- matlines[rep(1:nr,rep(2,nr))+c(0,nr),]
    segments3d(matlines)
  }
}
```

Here is a direct application of this method with four graphical illustrations for four different viewing angles.

```
> require("rgl")
> plot3d(reg2)
```


x1
x2
y
-4
-2
0
2
4
6
50
0
-50
-100
40
20
0
-20
-40