Writing better R code

Laurent Gatto*

June 26, 2013

Contents

More material is available at https://github.com/lgatto/TeachingMaterial.

*lg390@cam.ac.uk
[1]http://creativecommons.org/licenses/by-sa/3.0/

1 Introduction

This section focuses on better `R` programming in terms of cleaner and elegant syntax and performance improvements in terms of identifying regions that deserve optimisation as well as simple parallelisation.

The `R` language is in many ways a *functional programming* language, although other programming paradigms are also available. Functions are essential parts of the language that can be passed as arguments to other functions or returned as function output. Writing functions is very simple and represents a very accessible way of abstraction.

2 Programming

Writing functions

An `R` function is composed by

- A **name** that will be used to call the function (but see anonymous functions later in section 2); in the code chunk below, we call our function `myFun`.
- A set on input formal **arguments**, that are defined in the parenthesis of the `function` constructor. The `myFun` example has two arguments, called `i` and `j`. It is possible to provide default values to arguments, as illustrated for `j`.
- A function **body**, with curly brackets (only the body is composed or a single expression).
- A **return** statement, that represents the output of the function. If no explicit `return` statement is provided, the last statement of the function is return by default. Functions only support single value, i.e. `return(i, j)` is an error. To return multiple values one needs to return a `list` of the respective return variables like `return(list(i, j))`.

```
> myFun <- function(i, j = 1) {
+     mn <- min(i, j)
+     mx <- max(i, j)
+     k <- rnorm(ceiling(i * j))
+     return(k[k > mn/mx])
+ }
> myFun(1.75, 4.45)
```

```
[1] 1.5953 0.4874 0.7383

> myFun(1.75) ## j = 1 by default

[1] 0.5758
```

The example below illustrate pass-by-copy semantics and scoping in `R`. `f1` shows that functions act on copies of their arguments, leaving the original variables intact.

```
> x <- 1
> f1 <- function(x) {
+     x <- x + 10
+     x
+ }
>
> f1(x)

[1] 11

> x ## unchanged

[1] 1
```

`f2` demonstrates that functions however have access the variables defined outside of their body (global variables), while still keeping then unmodified.

```
> f2 <- function() {
+     x <- x + 10
+     x
+ }
>
> f2()

[1] 11

> x ## still unchanged

[1] 1
```

Function to create functions

Functions can also be written that generate new functions.

```
> make.power <- function(n)
+     function(x) x^n
> square <- make.power(2)
> cube <- make.power(3)
```

```
> square

function(x) x^n
<environment: 0x3135ba8>

> get("n", environment(square))

[1] 2

> square(2)

[1] 4

> cube(2)

[1] 8
```

Another interesting example is the `colorRampPalette` function that, given a vector of valid colour characters as input, returns a function that will create a colour palette along the initial colours.

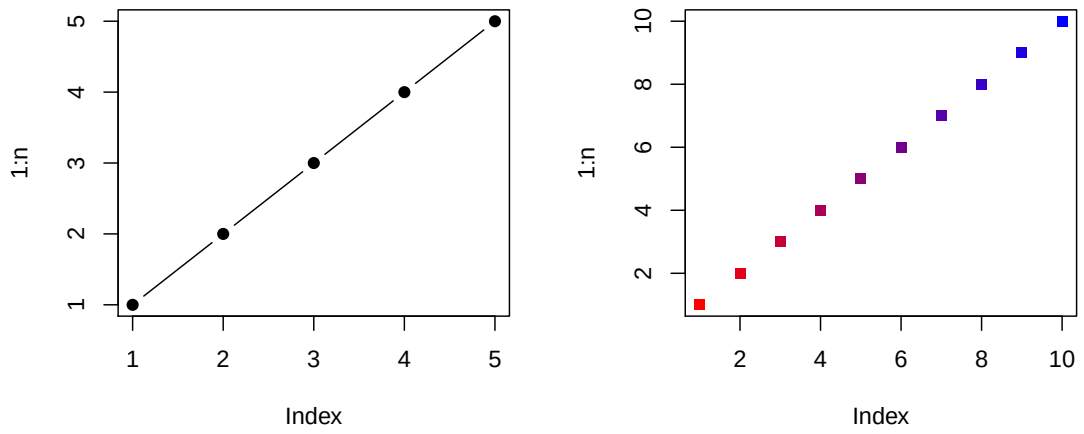
```
> (rbramp <- colorRampPalette(c("red", "blue")))

function (n)
{
    x <- ramp(seq.int(0, 1, length.out = n))
    rgb(x[, 1L], x[, 2L], x[, 3L], maxColorValue = 255)
}
<bytecode: 0x333b628>
<environment: 0x3323b28>

> rbramp(3)

[1] "#FF0000" "#7F007F" "#0000FF"

> rbramp(7)

[1] "#FF0000" "#D4002A" "#AA0055" "#7F007F" "#5500AA"
[6] "#2A00D4" "#0000FF"
```

The ... arguments

When an arbitrary number of arguments is to be passed to a function or of some arguments need to be passed down to an inner function, one can use the ... special arguments.

```
> plt <- function(n, ...)
+     plot(1:n, ...)
```

```
> par(mfrow = c(1, 2))
> plt(5, pch = 19, type = "b")
> plt(10, col = rbramp(10), pch = 15)
```




```
> args(cat)

function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
NULL

> args(rm)

function (..., list = character(), pos = -1, envir = as.environment(pos),
    inherits = FALSE)
NULL
```

Functions as arguments

Using functions to generate input to other function is quite natural in R : `sort(rnorm(5))` or `x[x > 0]`[2]. There is however a family of functions, the `*apply` functions, that are systematically called with other functions as arguments.

The general usage of three of the most apply members is as follows

- `lapply(X, FUN, ...)` iterates over each element of the `vector` or `list` `X` and applies function `FUN` to return a `list` of same length than `X`. Each element of the returned list is the return value of `FUN` for that respective element of `X`. Additional arguments can be passed to `FUN` through `...`.

```
> lapply(1:2, rnorm)

[[1]]
[1] 1.512

[[2]]
[1]  0.3898 -0.6212

> lapply(1:2, rnorm, 10, 2)

[[1]]
[1] 5.571

[[2]]
[1] 12.25  9.91
```

- `sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)` is a wrapper around `lapply` and returns an `vector`, `matrix` or `array` (if possible).

```
> library(fortunes)
> lapply(sample(315, 1), fortune)

[[1]]

Tom Backer Johnsen: I have just started looking at R,
and are getting more and more irritated at myself for
```

[2] `x > 0` is syntactic sugar for `` `>`(x, 0) `` and `x[i]` is in fact `` `[`(x, i) ``. As such, `x[x > 0]` can be rewritten `` `[`(x, `>`(x, 0)) ``, where the `>` function is used as an argument to the `[` function.

```
not having done that before. However, one of the
things I have not found in the documentation is some
way of preparing output from R for convenient
formatting into something like MS Word.
Barry Rowlingson: Well whatever you do, don't start
looking at LaTeX, because that will get you even more
irritated at yourself for not having done it before.
   -- Tom Backer Johnsen and Barry Rowlingson
      R-help (February 2006)

> sapply(sample(315, 1), fortune)

        [,1]
quote   "Let's not kid ourselves: the most widely used piece of software for s
author  "Brian D. Ripley"
context "'Statistical Methods Need Software: A View of Statistical Computing'"
source  "Opening lecture RSS 2002, Plymouth"
date    "September 2002"
```

- `apply(X, MARGIN, FUN, ...)` iterates of `MARGIN` of `array X`, apply function `FUN` and return the corresponding `vector`, `array` or `list`, depending on the return value of `FUN`.

```
> set.seed(10)
> m <- matrix(rnorm(10), ncol = 2)
> apply(m, 1, myFun)

       [,1]    [,2]    [,3]    [,4]   [,5]
[1,] 1.1018 -0.2382 0.74139 -0.9549 0.9255
[2,] 0.7558  0.9874 0.08935 -0.1952 0.4830

> apply(m, 1, myFun)

[[1]]
numeric(0)

[[2]]
[1] -0.6749
```

```
[[3]]
[1] -1.2652 -0.3737

[[4]]
[1] -0.6876 -0.8722

[[5]]
[1] -0.1018 -0.2538

> apply(m, 1, max) ## Biobase::rowMax

[1]  0.3898 -0.1843 -0.3637 -0.5992  0.2945

> apply(m, 2, min) ## Biobse::rowMin

[1] -1.371 -1.627
```

- `mapply(FUN, ...)` applies `FUN` to the first elements of each `...` argument, then the second elements, and so on.

```
> mapply(rep, 1:4, 4:1)

[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

The `replicate` function is a wrapper around `sapply` and allows repeated evaluation of a function call. See section 3 for an example.

Anonymous functions

Sometimes, when using `apply` for example, there is not save function to plug-in directly available and the operation to be performed is a one off. Instead of explicitly creating a function as shown in section 2, one tends to create an *anonymous* function, i.e. create a function on the fly without explicitly assigning it to a name.

```
> m

         [,1]    [,2]
[1,]  0.01875  0.3898
[2,] -0.18425 -1.2081
[3,] -1.37133 -0.3637
[4,] -0.59917 -1.6267
[5,]  0.29455 -0.2565

> apply(m, 1, function(x) ifelse(mean(x) > 0, mean(x), max(x)))

[1]  0.20427 -0.18425 -0.36368 -0.59917  0.01903
```

3 Performance

When performance becomes critical or the code becomes remarkably slow and it becomes necessary to improve performance, it is essential to start by assessing what portions of the code are to be optimised. It is also often useful to compare timings of two approaches and compare implementations.

Measuring execution time

We will be using the `system.time` function to compare `for` loops with and without initialisation and the `apply` functions while iterating over the elements of a `list` of length `N` = 10^4. We will then compute the mean of each element of the list and multiply it by its length as implemented in function `f`.

```
> ll <- lapply(sample(N), rnorm)
> f <- function(x) mean(x) * length(x)
```

The first approach we want to test is to use a `for` loop and append the results at the end if a vector `res1`. The important point in the first example is that `res1` is grown dynamically at each iteration.

```
> res1 <- c()
> system.time({
+     for (i in 1:length(ll))
+         res1[i] <- f(ll[[i]])
+ })

   user  system elapsed
 11.188   1.584  12.831
```

In the second example, we will use the same for loop but the result vector `res2` is initialised and the respective element set throughout the iterations.

```
> res2 <- numeric(length(ll))
> system.time({
+     for (i in 1:length(ll))
+         res2[i] <- f(ll[[i]])
+ })

   user  system elapsed
  0.824   0.000   0.826
```

The last approach uses the `sapply` idiom to apply `f` over each element of the list to generate the resulting `res3` vector.

```
> system.time(res3 <- sapply(ll, f))

   user  system elapsed
 11.572   1.772  13.423
```

The first approach is the slowest one, and the difference would become more substantially more pronounced for increasing values of `N`. This is because at each i^{th} iteration, when a new result of `f` is appended to `res1`, a copy of `res1` of length $i-1$ is generated to be appended the i^{th} results, essentially resulting in the duplication of long (and longer) temporary lists. This can be easily avoided by properly initialising the result vector `res2` or by using the `sapply` function, that will take care of the housekeeping for us.

Note that in general, using `apply` is not faster than a for loop with proper initialisation. It is however important to appreciate the conciseness and elegance of the last solution.

From the example above, we can hardly conclude that any of solutions 2 or 3 are faster than the other one, as we do not have any estimate of the variability of the

timing (which is rather sad, using an environment for statistical computing). It is very easy to obtain such an estimation by replicating the call, which is elegantly done using the `replicate` function.

```
> summary(replicate(50, system.time(res3 <- sapply(ll, f))["elapsed"]))

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.711   0.718   0.723   0.725   0.730   0.769
```

At this stage, we can't actually conclude anything as we have not verified that our three solutions produce identical results. This will be the topic of section 4.

Benchmarking

A more thorough benchmarking can be done using one the the rbenchmark or microbenchmark packages. Let's embed solutions 2 and 3 in two functions `sol2` and `sol3` to facilitate the direct comparisons.

```
> sol2 <- function(x) {
+     n <- length(x)
+     ans <- numeric(n)
+     for (i in 1:n) {
+         ans[i] <- f(x[[i]])
+     }
+     ans
+ }
> sol3 <- function(x)
+     sapply(x, f)
```

```
> library("microbenchmark")
> microbenchmark(sol2(ll), sol3(ll), times = 200)

Unit: milliseconds
     expr   min    lq median    uq   max neval
 sol2(ll) 791.9 806.0  812.1 821.5 932.3   200
 sol3(ll) 711.3 727.9  733.0 740.1 809.0   200
```

Based on the benchmarking above, we can now conclude that solution 3 using `sapply` is, under these conditions, faster. This can however not be generalised for all for (with initialisation) and `*apply` comparisons.

Profiling

To conclude this section on measuring performance, we introduce the `Rprof` function, that allows a detailed and complete time profiling. Its usage is simple. The user initiated the profiling by calling `Rprof()` (optionally passing a custom file name as input). From now on, every function call is going to be timed until profiling is switched of with `Rprof(NULL)`.

```
> Rprof()
> tmp <- replicate(10, sol3(ll))
> Rprof(NULL)
```

The detailed report can now be produced using the `summaryRprof` function (optionally specifying the file storing the profiling timings).

```
> summaryRprof()

$by.self
                  self.time self.pct total.time total.pct
"lazyLoadDBfetch"     10.98    59.54      10.98     59.54
"mean.default"         4.70    25.49       4.70     25.49
"mean"                 1.90    10.30       6.60     35.79
"FUN"                  0.62     3.36      18.42     99.89
"lapply"               0.12     0.65      18.42     99.89
"*"                    0.06     0.33       0.06      0.33
"array"                0.02     0.11       0.02      0.11
"length"               0.02     0.11       0.02      0.11
"unique.default"       0.02     0.11       0.02      0.11

$by.total
                    total.time total.pct self.time self.pct
"<Anonymous>"            18.44    100.00      0.00     0.00
"block_exec"             18.44    100.00      0.00     0.00
"call_block"             18.44    100.00      0.00     0.00
"doTryCatch"             18.44    100.00      0.00     0.00
"eval"                   18.44    100.00      0.00     0.00
"evaluate"               18.44    100.00      0.00     0.00
"evaluate_call"          18.44    100.00      0.00     0.00
"handle"                 18.44    100.00      0.00     0.00
"in_dir"                 18.44    100.00      0.00     0.00
"knit"                   18.44    100.00      0.00     0.00
"knit2pdf"               18.44    100.00      0.00     0.00
```

```
"process_file"             18.44    100.00      0.00     0.00
"process_group.block"      18.44    100.00      0.00     0.00
"replicate"                18.44    100.00      0.00     0.00
"sapply"                   18.44    100.00      0.00     0.00
"try"                      18.44    100.00      0.00     0.00
"tryCatch"                 18.44    100.00      0.00     0.00
"tryCatchList"             18.44    100.00      0.00     0.00
"tryCatchOne"              18.44    100.00      0.00     0.00
"withCallingHandlers"      18.44    100.00      0.00     0.00
"withVisible"              18.44    100.00      0.00     0.00
"FUN"                      18.42     99.89      0.62     3.36
"lapply"                   18.42     99.89      0.12     0.65
"sol3"                     18.42     99.89      0.00     0.00
"lazyLoadDBfetch"          10.98     59.54     10.98    59.54
"mean"                      6.60     35.79      1.90    10.30
"mean.default"              4.70     25.49      4.70    25.49
"simplify2array"            0.28      1.52      0.00     0.00
"unique"                    0.26      1.41      0.00     0.00
"unlist"                    0.24      1.30      0.00     0.00
"*"                         0.06      0.33      0.06     0.33
"array"                     0.02      0.11      0.02     0.11
"length"                    0.02      0.11      0.02     0.11
"unique.default"            0.02      0.11      0.02     0.11

$sample.interval
[1] 0.02

$sampling.time
[1] 18.44
```

4 Testing

As mentioned above when comparing for and `sapply` timings, for our comparison to make sense, we must first verify that our results are correct, i.e. in this case that our alternative implementations produce identical results.

The best way to verity exact equality between two arbitrary objects is to use the `identical` comparator.

```
> identical(res1, res2)

[1] TRUE
```

To test for identity of 3 objects, we propose function `identical3`, taken from the Matrix package[3].

```
> identical3 <-
+     function(x,y,z) identical(x,y) && identical (y,z)
> identical3(res1, res2, res3)

[1] TRUE
```

Sometimes, exact identity is not desired. A well known example (see R FAQ 7.31[4] for details) is the following.

```
> x <- sqrt(2)
> x * x == 2

[1] FALSE

> identical(x*x, 2)

[1] FALSE
```

Because floating numbers can not be represented exactly in a computer, one needs to limit the precision of the comparison. Instead of manually rounding the values to be compared, one can use the hardware specific tolerance `.Machine$double.eps` for this, and in particular the `all.equal` function.

```
> all.equal(x * x, 2)

[1] TRUE
```

The above illustrate that when specific expectations are to be met (whether on results or directly on function inputs), it is advisable to explicitly test them using the appropriate comparison operator. In particular, the `stopifnot` function provides a simple idiom for such testing.

[3] See in the `"test-tools-1.R"` from the Matrix package for other similar clever testing functions.
[4] Why doesn't R think these numbers are equal? http://www.hep.by/gnu/r-patched/r-faq/R-FAQ_82.html

```
> stopifnot(x * x == 2)

Error:  x * x == 2 is not TRUE

> stopifnot(all.equal(x * x, 2))
```

RUnit and testthat package are two packages that provide a more general framework for testing, in particular for unit testing in the frame of package development.

5 Parallelisation

The parallel package.

Other considerations: load balancing.

6 Debugging

An example illustrating - traceback - debug/undebug - fix - options(error = recover)

Cite R. Peng's debugging tutorial.

7 Further reading

This document presents an overview of certain useful idioms that are used in R. Other interesting topics related to R programming and suggested reading are package development (see for example QuickPackage or R package development) and object-oriented (see for example Short S4 tutorial or R object oriented programming), available on the repository.

Session information

All software and respective versions used to produce this document are listed below.

- R Under development (unstable) (2013-06-16 r62969), x86_64-unknown-linux-gnu
- Locale: LC_CTYPE=en_GB.UTF-8, LC_NUMERIC=C, LC_TIME=en_GB.UTF-8, LC_COLLATE=en_GB.UTF-8, LC_MONETARY=en_GB.UTF-8, LC_MESSAGES=en_GB.UTF-8, LC_PAPER=C, LC_NAME=C, LC_ADDRESS=C, LC_TELEPHONE=C, LC_MEASUREMENT=en_GB.UTF-8, LC_IDENTIFICATION=C
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils

- Other packages: fortunes 1.5-0, knitr 1.2
- Loaded via a namespace (and not attached): digest 0.6.3, evaluate 0.4.3, formatR 0.7, stringr 0.6.2, tools 3.1.0