Writing better R code

Laurent Gatto*

January 3, 2014

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, code profiling and testing, performance improvements via parallelisation and debugging. These topics are not covered in details but are presented as practical use-cases.

2 Programming

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.

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 of 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 (required only the body is composed of multiple expressions).
- 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 return value, i.e. `return(i, j)` is an error. To return multiple values one needs to return a `list` of the respective 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: 0x314e978>

> 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))
    if (ncol(x) == 4L)
        rgb(x[, 1L], x[, 2L], x[, 3L], x[, 4L], maxColorValue = 255)
    else rgb(x[, 1L], x[, 2L], x[, 3L], maxColorValue = 255)
}
<bytecode: 0x23d40a8>
<environment: 0x23d6c38>

> rbramp(3)
```

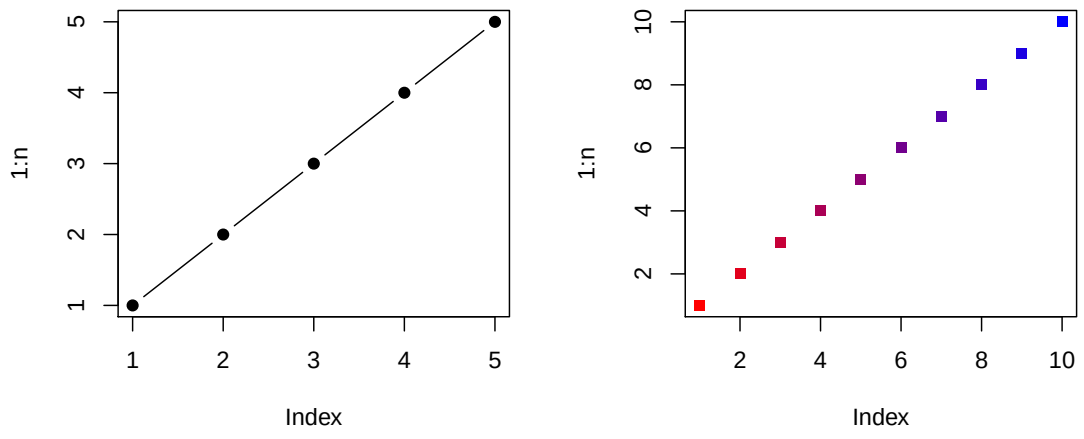
```
[1] "#FF0000" "#7F007F" "#0000FF"

> 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 if 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 illustrated below.

`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)
```

[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.

```
[[1]]

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

`tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)` applies function `FUN` to each group of values of the atomic object `X`; the groups `INDEX` are defined as a list of one or more factors.

```
> dfr <- data.frame(f1 = sample(LETTERS[1:2], 10, replace = TRUE),
+                   f2 = sample(LETTERS[3:4], 10, replace = TRUE),
+                   x = rnorm(10))
> tapply(dfr$x, dfr$f1, mean)

         A          B
 0.0008889 -0.2546474

> tapply(dfr$x, dfr$f2, mean)

       C        D
 0.06581 -0.21735

> tapply(dfr$x, list(dfr$f1, dfr$f2), mean)

        C        D
A  0.3547 -0.26449
B -0.3676 -0.02882
```

See also `by` or `split` for similar behaviours.

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
  1.060   0.124   1.187
```

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
  0.784   0.000   0.792
```

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
  0.708   0.000   0.718
```

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.
  0.697   0.710   0.755   0.755   0.787   0.937
```

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) 747.7 764.8  775.3 823.6 1157.5   200
 sol3(ll) 693.9 710.5  724.6 777.2  905.3   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("sol3.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("sol3.Rprof")

$by.self
               self.time self.pct total.time total.pct
```

```
"mean.default"      4.60    63.36       4.60     63.36
"mean"              1.94    26.72       6.54     90.08
"FUN"               0.38     5.23       7.26    100.00
"lapply"            0.22     3.03       7.26    100.00
"*"                 0.06     0.83       0.06      0.83
"unlist"            0.04     0.55       0.18      2.48
"length"            0.02     0.28       0.02      0.28

$by.total
                total.time total.pct self.time self.pct
"FUN"                 7.26    100.00      0.38     5.23
"lapply"              7.26    100.00      0.22     3.03
"replicate"           7.26    100.00      0.00     0.00
"sapply"              7.26    100.00      0.00     0.00
"sol3"                7.26    100.00      0.00     0.00
"mean"                6.54     90.08      1.94    26.72
"mean.default"        4.60     63.36      4.60    63.36
"unlist"              0.18      2.48      0.04     0.55
"simplify2array"      0.18      2.48      0.00     0.00
"unique"              0.16      2.20      0.00     0.00
"*"                   0.06      0.83      0.06     0.83
"length"              0.02      0.28      0.02     0.28

$sample.interval
[1] 0.02

$sampling.time
[1] 7.26
```

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` compactor.

```
> 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

```
> 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.

```
> 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.

[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

5 Parallelisation

In addition to the elegant syntax of the `apply` family of functions, an additional advantage is that due their underlying iterative nature, where the same function is called independently on each element of the list/vector/array, it is an obvious candidate for parallelisation. The parallelisation support is provided by the parallel,that is one of the recommended packages that are shipped with `R` since version 2.14.

Let's first illustrate the parallelisation using the `mclapply` function; it is a parallel version of `lapply` (see section 2). We pass it an anonymous function that returns the process id using and specify the number of cores to use with the `mc.cores` argument. The `detectCores` function attempts to detect the number pf CPU cores on the host.

```
> library("parallel")
> detectCores()

[1] 4

> mclapply(1:3, function(x) Sys.getpid(), mc.cores = 3)

[[1]]
[1] 3273

[[2]]
[1] 3274

[[3]]
[1] 3275

> mclapply(1:3, function(x) Sys.getpid(), mc.cores = 2)

[[1]]
[1] 3276

[[2]]
[1] 3277

[[3]]
[1] 3276
```

We see that when specifying 3 computations on 3 cores (out of 4 possible), our

construct returns 3 different process identifiers. When parallelising the same task on 2 available cores, one process executes two of the computations.

The application of such parallelisation first tasks where *independent* computations are repeated certain number of times; results just need to be combined after parallel executions are done. In this section, we consider two frameworks.

- A cluster of nodes (as in package snow): generate multiple workers listening to the master; these workers are new processes that can run on the current machine or on similar ones with an identical `R` installation. This framework should work on all `R` platforms.

  This approach needs to explicitly create the cluster nodes using the `makeCluster` function. The cluster of nodes can be stopped with the `stopCluster` functions. An important aspect of this solution is that all symbols used within the parallelised functions must be exported to each node of the cluster. This is achieved with the `clusterExport`.

- The `R` process is *forked* to create new `R` processes by taking a complete copy of the masters process, including the workspace (pioneered by package multicore). This does not work on Windows though and the parallel function will fall back in serialised execution (`mc.cores = 1`).

Let now build a parallelised solution for the example of section 3.

```
> solmc <- function(x)
+     mclapply(x, f)
> solpar <- function(x, cl)
+     parLapply(cl, x, f)
> sol3 <- function(x)
+     lapply(x, f)
> cl <- makeCluster(4)
> stopifnot(identical3(sol3(ll), solmc(ll), solpar(ll, cl)))
> stopCluster(cl)
```

If we were to compare the speed to these 3 implementations, we would observe that the parallelised version would hardly beat the serialised implementation, and most likely be slower due to the overhead of the parallelisation and the light computing task of the example. Instead, we have benchmarked the following code.

```
library("parallel")
library("microbenchmark")
ll <- replicate(8, matrix(rnorm(1e6),1000), simplify=FALSE)
f <- function(x) mean(solve(x), trim=0.7)
pbench <- microbenchmark(
    res <- lapply(ll, f),
    resmc <- mclapply(ll, f, mc.cores = 16L),
    times = 10)
stopifnot(identical(res, resmc))
save(pbench, file = "pbench.rda")
```

```
Unit: seconds
                                      expr   min    lq
                    res <- lapply(ll, f) 4.978 5.054
 resmc <- mclapply(ll, f, mc.cores = 16L) 1.491 1.612
 median    uq   max neval
  5.235 5.434 5.497    10
  1.656 1.701 1.809    10
```

Several parameters will influence how much can be gained by parallelising code as illustrated above, including number of cores and overhead of the parallelisation, speed of individual calculations, possibly other limiting factors like disk access or load balancing (when when several of the jobs to be run in parallel take different times).

Finally, other frameworks exists, like the foreach package and Bioconductor's BiocParallel package (currently still in development).

Further reading The *Parallel R* book by McCallum and Weston, O'Reilly (2011), the parallel and foreach vignettes and the *High Performance Computing* CRAN task view[5].

6 Debugging

Let's consider the following case[6] where executing function g produces and error which is not directly results from g's code, but from another function called, possibly indirectly, inside g.

[5] http://cran.r-project.org/web/views/HighPerformanceComputing.html
[6] Example taken from slides by Martin Morgan and Robert Gentleman.

```
> g()
Error in x[-1:2] (from #3) : only 0's may be mixed with negative subscripts
> g
function() f()
```

The first step is to isolate the function in which the error is thrown. This can be achieved with the `traceback` function, that will print the call stack of the last call error.

```
> traceback()
5: FUN(1:10[[5L]], ...)
4: lapply(X = X, FUN = FUN, ...)
3: sapply(1:10, e) at #1
2: f() at #1
1: g()
```

We see that after calling `g` manually, `f` was called, then `sapply` iterated over `1:10`, called function `e` at each iteration. `sapply` is a wrapper around `lapply`, which we see in position 4 and `FUN`, i.e. `e` fails at the fifth position on the stack with the error

```
Error in x[-1:2] (from #3) : only 0's may be mixed with negative subscripts
```

From here on, one can display the code the the offending function `e` and even reproduce the error directly.

```
e
function(i) {
  x <- 1:4
  if (i < 5) x[1:2]
  else x[-1:2]
}
e(5)
Error in x[-1:2] (from #3) : only 0's may be mixed with negative subscripts
```

The next step is typically to the fault function for debugging using with `debug(e)`, so that `browser()` will be called on entry. In `browser` mode, the execution of an expression is interrupted and it is possible to inspect the state of the environment and the content of the respective variables. Stepping to the next line is done by pressing `enter` of `n`. Typing `Q` can be used to exit debugging. To unregister the faulty function from debugger mode, use `undebug(e)`.

```
> debug(e)
> e(5)
debugging in: e(5)
debug at #1: {
    x <- 1:4
    if (i < 5)
        x[1:2]
    else x[-1:2]
}
Browse[2]>
debug at #2: x <- 1:4
Browse[2]>
debug at #3: if (i < 5) x[1:2] else x[-1:2]
Browse[2]> ls()
[1] "i" "x"
Browse[2]> i
[1] 5
Browse[2]> x
[1] 1 2 3 4
Browse[2]>
debug at #3: x[-1:2]
Browse[2]> x[-1:2]
Error in x[-1:2] (from #3) : only 0's may be mixed with negative subscripts
Browse[2]> x[-(1:2)]
[1] 3 4
Browse[2]> Q
> undebug(e)
> fix(e)
```

Finally, once the error has been identified, it is possible to fix the bug immediately with `fix(e)`, which will open the default editor for the user to made the necessary changes. After saving and closing, a new copy of the function will be available in the global environment.

Other tools of interest are `trace()` to insert code into functions. It is also possible to set `options(error=recover)` to get the call stack and enter into browser mode in any of the function calls.

Further reading *An Introduction to the Interactive Debugging Tools in R*[7] by Roger Peng, the *Debugging `R` code* slides[8] by Gatto and Stojnić and the *Debugging* section[9] of the *Writing R Extensions* manual. Several editor have debugging facilities, including the `StaET` eclipse plugin[10] and `emacs`' `ess tracebug`[11].

7 Other topics of interest

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) (2013-10-16 r64064), `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=en_GB.UTF-8`, `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, parallel, stats, utils
- Other packages: fortunes 1.5-0, knitr 1.5
- Loaded via a namespace (and not attached): digest 0.6.3, evaluate 0.5.1, formatR 0.10, highr 0.3, microbenchmark 1.3-0, stringr 0.6.2, tools 3.1.0

[7] http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf
[8] https://github.com/lgatto/R-debugging
[9] http://cran.r-project.org/doc/manuals/r-release/R-exts.html#Debugging
[10] http://www.walware.de/goto/statet
[11] http://ess.r-project.org/Manual/ess.html#ESS-tracebug