Enhancing your research with R

Klaus Nordhausen

17.10.2018

CSTAT - Computational Statistics
Institute of Statistics & Mathematical Methods in Economics
Vienna University of Technology

Introduction

Why the need to produce good code?

- Generally statisticians work with data which need to be handled somehow.
- When writing applied research papers, the analysis must be reproducible.
- In more theoretical statistics, new methods have to be implemented, verified and compared to competing methods in simulation studies, again everything must be reproducible.
- More and more journals want (insist) to publish the code which reproduces all results of the paper with the article.

Therefore as a researcher in statistics one needs to be able to produce code which is presentable, comprehensibly and efficient.

Also, your methods are much more likely to be used in practice if code is publicly available.

Ideal case

In an ideal case one could image:

1. Write a great theoretic paper suggesting a new method.
2. Publish the code of the new method.
3. Write a paper about the code (there are many outlets here, like Journal of Statistical Software, R Journal, Journal of Open Source Software,...)
4. Write an applied paper showing how your method works for real data.

Which software to take?

Naturally, many languages are available which can be used to implement the new methods.

Currently R can be considered the lingua franca of statistics which makes it a likely choice as it is used by many users, the majority of competing methods will be available in R and R has a great infrastructure to share code.

Code can be shared for example as:

1. just as a script.
2. an R package which can then be available on your homepage, GitHub, CRAN, R-Forge, Bioconductor, ...
3. a Shiny app online.

The rest of the talk will assume thus that the choice is R.

General R coding recommendations

Top-down programming

General agreement is that good code is written in a modular manner. This means when you have to implement a procedure, you decompose it into small parts where each part will become an own function.

Then the main function is “short” and will consist mainly of calling these subfunctions. Naturally also within these functions the same approach should be taken.

Good practise

Good practice is also to

- use consistent naming conventions (i.e. `snake_case`, `CamelCase`,...) and use meaningful object names. Avoid names which are already used in base R (`T`, `F`, `t`, `mean`, `data`, ...) and do not use names with a dot in between (`my.function`).
- comment your code in a way that others and you can still read it later.

Often the code is first written so that it just works and then optimized for speed and memory usage.

Debugging and profiling

Debugging

There are two commonly referred claims:

1. Programmers spend more time on debugging their own code than actually programming it.
2. In every 20 lines of code is at least one bug.

Hence debugging is an essential part of programming and there are strategies and tools available in R to do this well.

Top-down debugging and small start strategy

The **top-down** strategy is also followed when debugging. First the top level function is debugged and all subfunctions are assumed to be correct. If this does not yield a solution, then the next level is debugged and so on.

The **small start** strategy in debugging suggests to start using small test cases for debugging.

Once these work fine, then consider larger testing cases.

At that stage also extreme cases should be tested.

R functions for debugging

R provides many functions helping in the debugging process. To name some:

- `browser`
- `debug` and `undebug`
- `debugger`
- `dump.frames`
- `recover`
- `trace` and `untrace`

For details about these functions see their help pages. In the following we will only look at `debug` and `traceback`.

Note that also RStudio offers special debugging tools, see `https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio` for details.

traceback

When using functions and errors occur, it is often not really obvious where the actual error occurred or which (sub)function caused the error.

One strategy then is to use the `traceback` function, which returns when called directly after the erroneous call the sequence of function calls, which lead to the error.

traceback II

```
> f1 <- function(x)  f2(x)^2
> f2 <- function(x)  log(x)+ "x"
> mainf <- function(x)
+   {
+   x <- f1(x)
+   y <- mean(x)
+   y
+   }
> mainf(1:3)
Error in log(x) + "x": non-numeric argument to binary operator
```

```
> traceback()
3: f2(x) at #1
2: f1(x) at #2
1: mainf(1:3)
```

debug

Assume you have a function `foo` you assume to be faulty. Using then

```
> debug(foo)
```

will open whenever the function is called the "browser" until either the function is changed or the debugging mode terminated using

```
> undebug(foo)
```

In the "browser" the function will be executed line by line where always the **next line to be executed** will be shown.

debug commands in browser mode

In the browsing mode the following commands have a special meaning:

- `n` (or just hitting enter) will execute the line shown and then present the next line to be executed.
- `c` this is almost like `n` just that it might execute several lines of code at once. For example if you are in a loop then `c` will jump to the next iteration of the loop.
- `where` this prints a stack trace, the sequence of function calls which led the execution to the current location
- `Q` this quits the browser.

In the browser mode any other R command can be used. However, to see for example the value of a variable `n`, the variable needs to be explicitly printed using `print(n)`.

Debugging demo

In a demo we will go through the following function in debugging mode

```
> SimuMeans <- function(m, n=100, seed=1)
+   {
+   set.seed(seed)
+
+   RES <- matrix(0, nrow=m, ncol=3)
+
+   for (i in 1:m){
+     X <- cbind(rnorm(n), rt(n,2), rexp(n))
+     for (j in 1:3){
+       RES[i,j] <- mean(X[,j])
+       }
+     print(paste(i,Sys.time()))
+     }
+   return(RES)
+   }
> debug(SimuMeans)
> SimuMeans(5)
```

Capturing errors

Especially in simulations it is often desired that if an error occurs, not the whole process is terminated but that the error is **catched** and an appropriate record made. Otherwise the simulations should continue.

R has for this purpose the function `try` and `tryCatch` where we will only consider `tryCatch`.

The idea of `tryCatch` is to run the "risky" part where errors might occur within the tryCatch call and tell `tryCatch` what to return in the case of an error.

Capturing errors demo

Consider a modified version of our previous simulation function:

```
> my.mean <- function(x){
+   na.fail(x)
+   mean(x)
+   }
> SimuMeans2 <- function(m, n=100, seed=1)
+   {
+   set.seed(seed)
+
+   RES <- matrix(0, nrow=m, ncol=3)
+
+   for (i in 1:m){
+     X <- cbind(rnorm(n), rt(n,2), rexp(n))
+     if (i==3) X[1,1] <- NA
+     for (j in 1:3){
+       RES[i,j] <- my.mean(X[,j])
+       }
+     #print(paste(i,Sys.time()))
+     }
+   return(RES)
+   }
> SimuMeans2(5)
Error in na.fail.default(x): missing values in object
```

Capturing errors demo II

Using tryCatch

```
> SimuMeans3 <- function(m, n=100, seed=1)
+   {
+   set.seed(seed)
+   RES <- matrix(0, nrow=m, ncol=3)
+   for (i in 1:m){
+     X <- cbind(rnorm(n), rt(n,2), rexp(n))
+     if (i==3) X[1,1] <- NA
+     for (j in 1:3){
+       RES[i,j] <- tryCatch(my.mean(X[,j]), error = function(e) NA)
+       }
+     #print(paste(i,Sys.time()))
+     }
+   return(RES)
+   }
> SimuMeans3(5)
            [,1]        [,2]      [,3]
[1,]  0.10888737 -0.29098535 1.1103408
[2,] -0.04920681 -0.17200178 0.8624404
[3,]          NA -0.02304963 1.0302238
[4,] -0.09209421 -0.27303215 1.0813623
[5,] -0.05374456  0.13526423 1.0200112
```

Profiling

If you know that your function is correct but think it is not very efficient you can do profiling which helps to identify the parts of the function which are bottlenecks and then you can consider if these parts could be improved.

The idea of profiling is that the software checks in very short intervals which function is currently used.

The main functions in R to do profiling are `Rprof` and `summaryRprof`. But there are also many other specialized packages for this purpose, like for example the package `profvis`.

A function to profile

```
> Stest <- function(n=1000000, seed=1)
+   {
+   set.seed(seed)
+   normals <- rnorm(n*10)
+   X <- matrix(normals, nrow=10)
+   Y <- matrix(normals, ncol=10)
+
+   XXt <-  X %*% t(X)
+   XXcp <- tcrossprod(X)
+
+   return(n)
+   }
> system.time(Stest())
   user  system elapsed
  16.22    0.69   19.77
```

A function to profile II

```
> Rprof(interval=0.01)
> Stest()
[1] 1e+06
> Rprof(NULL)
> summaryRprof()$by.self
              self.time self.pct total.time total.pct
"rnorm"           14.72    67.99      14.72     67.99
"%*%"              2.51    11.59       2.51     11.59
"matrix"           1.71     7.90       1.71      7.90
"tcrossprod"       1.55     7.16       1.55      7.16
"t.default"        1.03     4.76       1.03      4.76
"ls"               0.02     0.09       0.02      0.09
"setHook"          0.01     0.05       0.02      0.09
"any.dots"         0.01     0.05       0.01      0.05
"anyNA"            0.01     0.05       0.01      0.05
"codeBufCode"      0.01     0.05       0.01      0.05
"findCenvVar"      0.01     0.05       0.01      0.05
```

Using profvis for profiling

The profvis package yields interactive results displayed in html and therefore preferably done in RStudio.

```
> library(profvis)
> profvis({
+   Stest <- function(n=1000000, seed=1){
+     set.seed(seed)
+     normals <- rnorm(n*10)
+     X <- matrix(normals, nrow=10)
+     Y <- matrix(normals, ncol=10)
+     XXt <-  X %*% t(X)
+     XXcp <- tcrossprod(X)
+     return(n)
+   }
+   Stest()
+ })
```

Using profvis for profiling II

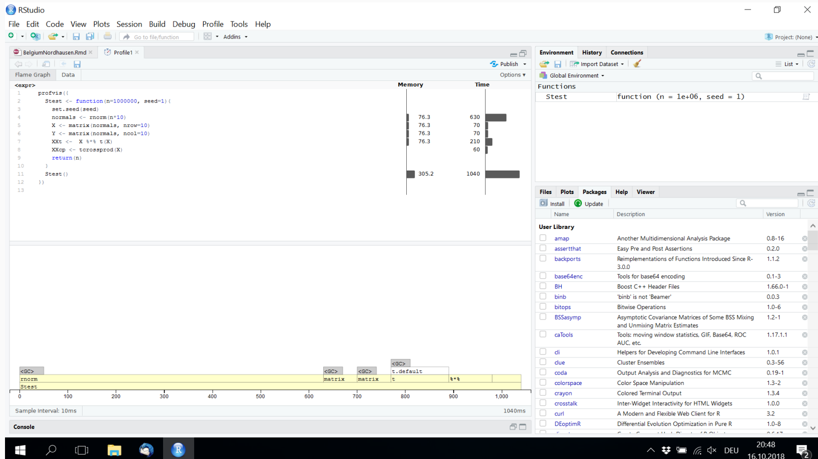

Figure 1:

Speeding up R code

Making code more efficient

Assuming the bottlenecks are identified. Then there are several options how to make the code more efficient.

- Improve the R code. E.g. by making more use of vectorization, `lapply` and friends or using the best function for the job and including setting function arguments properly. For example `eigen(cov(x), symmetric = TRUE)` instead of `eigen(cov(x))` or `crossprod(X,Y)` instead of `t(X) %*% Y`.
- Parallelize your code if possible.
- Write for the slow part code using C, C++ or FORTRAN which can be easily called then from within R.

Parallelizing code in R

Parallelization does not really speed up computations time, but it divides the work to several **workers** and can save therefore our time.

R offers many possibilities to use parallel computing, for an overview see for example the CRAN TASK View High Performance Computing `https://cran.r-project.org/web/views/HighPerformanceComputing.html`.

In general paralellization in R is easier on Linux or Mac but there are also approaches which work an all operating systems.

Emabarassingly parallel problems

Many uses of `lapply` and friends and ‘for loops perform computations independently form each other and are therefore easily parallelized.

The package `parallel` offers here many functions to work almost the same way as the ‘’normal” function in base R. For example on linux and Mac `lapply` can be simply replaced by `mclapply` and the only needed change is that one has to specify the number of cores to be used.

Parallel computing of bootstrapping

Using bootstrapping is often quite simple in base R using the function `replicate`:

```
> set.seed(1)
> x <- rnorm(50)
> m <- 200
> n <- length(x)
> test_statistic <- function(x, mu=0) { sqrt(length(x))*
+     (mean(x)-mu) / sd(x)}
> xNull <- x-mean(x)+0
> boot_statistics <- replicate(m, test_statistic(sample(xNull,
+                      n, replace = TRUE)))
> (sum(abs(test_statistic(x)) < abs(boot_statistics)) + 1)/
+   (m + 1)
[1] 0.4079602
> t.test(x)$p.value
[1] 0.3970852
```

Parallel computing for bootstrapping II

```
> library(parallel)
> ncores <- 3
> cl <- makeCluster(ncores, type = "PSOCK")
> clusterExport(cl, c("xNull", "m","n", "test_statistic"))
> clusterSetRNGStream(cl = cl, iseed = 123)
> boot_statistics_p <- parSapply(cl, 1:m, function(i, ...){
+             test_statistic(sample(xNull, n, replace = TRUE))
+         })
> stopCluster(cl)
> (sum(abs(test_statistic(x)) < abs(boot_statistics_p))
+   + 1)/(m + 1)
[1] 0.4278607
```

Rcpp

The Rcpp package makes it especially easy to combine R with C++ Code. There are then many additional RcppXXX packages which provide many useful additional C++ features. Like for example RcppEigen or RcppArmadillo which provide classes for linear algebra.

Together with Rstudio the C++ code can then be easily run directly in R using in a cpp file a block

```
/*** R
R code
*/
```

and sourcing then the cpp file.

Simple Rcpp for linear regression

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;
// [[Rcpp::export]]
SEXP LMcpp(SEXP y, SEXP x)
{
  mat X = as<arma::mat>(x);
  vec Y = as<arma::vec>(y);
  int p = X.n_cols;
  int n = X.n_rows;

  arma::colvec coef = arma::solve(X, Y);
  arma::colvec resid = Y - X*coef;
  double sig2 = arma::as_scalar(arma::trans(resid)*resid/(n-p));

  return Rcpp::List::create(Rcpp::Named("coefficients") = coef,
                            Rcpp::Named("sigma2") = sig2,
                            Rcpp::Named("residuals") = resid
  );
}
```

Simple Rcpp for linear regression II

```
/*** R
set.seed(1)
n <- 1000
x1 <- runif(n)
x2 <- rbeta(n, 1, 0.5)
eps <- rnorm(n)
y <- 2 + x1 + 0.5 * x2 + eps
X <- cbind(1,x1,x2)
RES <- LMcpp(y,X)
RES[1]
lm.fit(X,y)[1]
*/
```

Simple Rcpp for linear regression III

```
RES[1]
$`coefficients`
          [,1]
[1,] 2.0339445
[2,] 0.8971072
[3,] 0.5834849

lm.fit(X,y)[1]
$`coefficients`
                 x1        x2
2.0339445 0.8971072 0.5834849
```

Making R packages

R packages

CRAN has over 12000 packages available for users. These additional "features" are maybe one reason for R's huge success.

Anyone can make an R package and publish it on CRAN if it just fulfills some formal criteria.

But making an own package is not only meaningful if you want to publish it worldwide. It might be also just for colleagues and collaborators in a project or even just for yourself as it gives clear structure to your work and makes it easy to understand even years later.

Making a package

The ultimate “rule book” for making an R package is “**Writing R Extensions**” which is part of every R installation.

There are however also many tutorials online and also RStudio is providing meanwhile an infrastructure to make R packages.

Making an R package is not difficult!

Package essentials

A basic package has the following structure:

1. a **name**
2. a description file called **description**
3. a namespace file called **namespace**
4. a folder with R code, called **R**
5. a folder containing documentation called **man**
6. a folder containing material to make a vignette, called **vignettes**
7. a folder containing the data called **data**
8. a folder for additional material like NEWS or CHANGELOG files called **inst**

An R package can still have more folders and actually only items 1.-5. are **mandatory**.

Automatic package structure creation

Given the name of the new package (in the following `DemoPackage`) and having all your functions ready, the easiest way to start a package is:

1. Start R with an empty workspace.
2. "Load" all the data sets and functions which should be included in the package into R. Assume for example they are `D1, f1, f2, f3`.
3. Change the working directory to the place where the package should go.
4. Use the function `package.skeleton` to create the first version of the package as follows:

```
> package.skeleton("DemoPackage", list=c("D1", "f1",
+                                        "f2", "F3"))
```

Automatic package structure creation II

The previous call will create in the working directory a folder with the folder name `DemoPackage` and the subfolders `R`, `man` and `data` and put the corresponding files into the subfolders. Also description and namespace files are created.

While the files in the `R` and `data` folders can usually stay as they are (if you have not made coding errors) all the other files are a first draft and all need still editing.

Also some extra files and folders might be needed to be created manually. In my opinion the hardest part is writing the help files.

Package building and checking

After all these steps, the package is basically ready. It needs only to be built and checked.

This can be done for example using the commands

```
R CMD build PackageName
```

If everything earlier was done correctly this should create the file `PackageName_version_number.tar.gz` which is the package source file.

```
R CMD check PackageName_version_number.tar.gz
```

Package building and checking II

Given your package passed all tests. It can be installed using

```
R CMD INSTALL PackageName_version_number.tar.gz
```

and making the windows binary version is built (on windows!) using

```
R CMD INSTALL --build PackageName_version_number.tar.gz
```

But building, checking and installation can also be made using RStudio.

Further information

1. **Writing R Extensions** contained in each R distribution or at `https://cran.r-project.org/doc/manuals/R-exts.html`
2. **R Packages** (Hadley Wickham) `http://r-pkgs.had.co.nz/`
3. **Creating R Packages: A Tutorial** (Friedrich Leisch) `http://cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf`
4. **Making an R Package** (R.M. Ripley) `http://portal.stats.ox.ac.uk/userdata/ruth/APTS2012/Rcourse10.pdf`

Other useful features in the context of R

No time to cover today, but also useful to be aware of are:

- CRAN TASK VIEWS
- RMarkdown
- Shiny

Happy coding!

Thank you for your attention!