**Introduction**

The result is **CPAT**, an R package implementing some change point analysis statistical tests. What **CPAT** does will be the subject of a future post (it will be published when the accompanying paper is made available online); what I want to focus on in this article is how I learned to organize an R research project, and how that culminated in CPAT.

**Executable vs. Package: Why Not Both?**

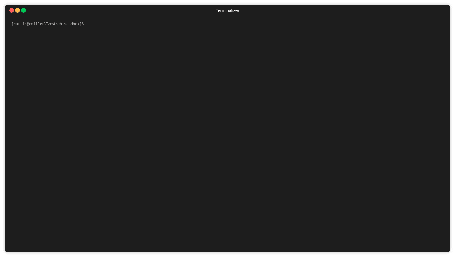
They are not at odds. **CPAT** demonstrates that it is possible to view an R project as both an executable and as a package. That said, the package development approach becomes dominant; making the package executable (from the command line) is an additional feature that makes the project even more portable and extensible.

If one is going to adopt the package development approach, one must use the hierarchy R packages needs. So that means:

* R code that defines the package (which are mostly just functions) is placed in the R/ directory.
* Project data goes in the data/ directory.
* Compiled code from other languages (such as C++ when using **Rcpp**) goes in the src/ directory.
* Long-form documentation goes in vignettes/. This could be the paper itself, if written in the form of a vignette.
* Other important files should be placed in a reasonably-organized inst/ directory, to be installed with the package, along with other files that should be installed into the base directory (such as Makefile). For example, I put all my plots in inst/plots/, and this would also be a good directory to put the paper that accompanies the project.
* Put executable scripts, including R scripts, in exec/.

The approach championed by Zelner doesn’t require a particular organizational style but simply that there be a coherent organization to the project. R package development not only has a coherent structure but even *enforces* it. If that structure doesn’t quite work, then one can add other files and directories as needed and note them in the .Rbuildignore file, so they’re ignored when the package is built.

When writing an R package, the relevant R tools basically enforce some essential points of style such as documenting objects. Also, the developer-researcher starts to think of important functionality of the project in terms of reusable functions that should be added to the package to be called by the scripts that actually execute the analysis—with documentation and everything else. Having well-documented functions, even if they serve a minor purpose, helps greatly in making the project more easily understood and written not only by others but by the original author as well.



While package development does place (helpful) constraints, it does not specify everything. In other words, there is room for style. I essentially define *style* to mean any aspect of programming in which a choice is made that was not determined by the programming language or software. Examples of style include naming conventions, indentation, etc. Consistent style makes for understandable code; having consistent style is arguably more important than the stylistic decisions made.

As I mentioned above, the package development approach turns out not to be mutually exclusive with the project-as-executable approach. While it seems like documentation on R package development (including Dr. Wickham’s book) mentions the exec/ directory of a package only in passing, I found it to be a good place to place executable R scripts. Similarly, make can still be used to automate analysis tasks; R packages allow for including make files.

So in addition to the files that essentially defined the package, I also wrote stand-alone, command line executable R scripts and placed them in the exec/ directory (which causes them to be flagged as “executable” when the package is installed). I wrote a Vim template file for R scripts that provides a skeleton for making the package executable from the command line. That template is listed below:

#!/usr/bin/Rscript

################################################################################

# MyFile.R

################################################################################

# 2018-12-31 (last modified date)

# John Doe (author)

################################################################################

# This is a one-line description of the file.

################################################################################

# optparse: A package for handling command line arguments

if (!suppressPackageStartupMessages(require("optparse"))) {

install.packages("optparse")

require("optparse")

}

################################################################################

# MAIN FUNCTION DEFINITION

################################################################################

main <- function(foo, bar, help = FALSE) {

# This function will be executed when the script is called from the command

# line; the help parameter does nothing, but is needed for do.call() to work

quit()

}

################################################################################

# INTERFACE SETUP

################################################################################

if (sys.nframe() == 0) {

cl\_args <- parse\_args(OptionParser(

description = "This is a template for executable R scripts.",

option\_list = list(

make\_option(c("--foo", "-f"), type = "integer", default = 0,

help = "A command-line argument"),

make\_option(c("--bar", "-b"), type = "character",

help = "Another command-line argument")

)

))

do.call(main, cl\_args)

}

Converting my scripts into modularized, executable programs was, not surprisingly, very time consuming, and the transition was not perfect; some scripts just could not be modularized well. Nevertheless, the end result was likely worth it, and I could then write a Makefile defining how the pieces fit together. This tamed the complexity of the project and made it more reproducible; someone looking to repeat my analysis should only have to type make in a Linux terminal[1](https://ntguardian.wordpress.com/2019/02/04/organizing-r-research-projects-cpat-case-study/#fn-3527-1) to see the results themselves.

**In Practice**

In some sense the end goal is to have an R package that could be distributed to others via, say, CRAN, so they can *use* the methods you employed and developed, not just reproduce your research; at least, that’s the case for me, a mathematical statistician interested in analyzing and developing statistical tests and procedures. When a package is written to contain research and not just for software distribution, it comes with a lot of files that aren’t needed for the package to function; just look at the dirctory tree!

The solution is to just delete the files that can be recreated—perhaps with make clean if you set it up right—and consider adding other files to .Rbuildignore when you want to distribute the package for others to use. So this isn’t actually a big problem.

I’m undecided whether this is good style. On the one hand, it’s nice that when others read your code there’s manual entries even for functions that are local to the project to further document what was done and how the code works. Even when distributing the software, having every function documented, even ones that are “private” to the package, seems to be in concordance with the spirit of open source software, making the source code easier understood by users who need and want to know how your software works. It also could serve as a good way to modularize documentation; a statistical formula is kept with the function that computes it rather than the interface to that function (which likely links to that underlying function). Having examples for those internal functions also should provide an additional layer of testing and helps when others want to extend the package.

On the other hand… most of the pages of the manual are devoted to functions the user isn’t supposed to be calling directly in their work. Of all those functions, maybe five are functions the user is expected to use. Should all that documentation space be devoted to something the user doesn’t use?

While I’m not set in my opinion, I lean to having more documentation rather than less, even if most of it is for private functions. After all, it’s useful to me when I’m developing the project and package.

**Conclusion**

I feel like spending those months to make my project logical and reporducible was time well spent. Not only did I learn a lot in the process. Additionally, this project is not over; my advisor and I are continuing to work on extending the results that lead to the creation of this package in the first place, which will call for more simulation experiments. Now that I’ve organized my work I now have a good base for continuing that work.

I hope that this article inspired others on how to organize their R research projects. Gauging from reactions to my previous article, I think this is an underappreciated topic, unfortunately. Having a plan for managing package complexity and organization goes a long way to keeping your work under control and helps others appreciate what you’ve done. It also can lead to your work having a greater impact since others can use it as well.

I got a lot of good feedback from my previous article. I look forward to hearing what the community has to say now. I’m always open to suggestion.