An Introduction to Developing R Packages

Women in Statistics and Data Science

October 20, 2017

Christina Knudson, Ph.D.
University of Saint Thomas

Lindsey Dietz, Ph.D.
Federal Reserve Bank of Minneapolis

Haema Nilakanta
University of Minnesota

We have produced this guide and the R package wisdom as our contribution to the Women in Statistics and Data Science conference. We hope to help guide women through the basic steps of R package production so that women's R functions can be packaged neatly and shared easily. The R package `wisdom` and the electronic (pdf) version of this guide with hyperlinks can be found at `https://github.com/knudson1/WSDS2017`.

1 Before Starting

1.1 Design Document

Before you even touch your computer, develop a clear vision of your project. Put this vision and all the nitty gritty details into a "design document." The design document is the document you will refer to every time you sit down to program your package. Ideally, your design document will be so thorough that you will only think about programming according to that document and you will not need to think about statistics or data science anymore. The design document is essential for long-term projects so that you can communicate to your future self. A design document is also essential if you are working on a team: it will ensure that one function can call another seamlessly and with no mismatching arguments, even if the functions are written by different people.

A design document should have an overview of the purpose of the package as well as all the details. Decide exactly what you want your package to do. Then, carefully plan out each function that you will include in your package. For each function,

- write down all equations that function relies on.
- decide the inputs and outputs, as well as the properties of each. For example, will the input be a matrix, vector, or scalar?
- decide whether you will provide any defaults for the arguments. If you will, select the defaults.
- write pseudocode and try to imagine if there is anything tricky you need to consider.

- decide whether it will be available to the user or not. You might write helper functions that will be called by other functions. Users might not see these helper functions, and you will not need to write documentation for them later.

We also highly recommend you brainstorm tests and include these in the design document. You want to test your code every step of the way to ensure every command and function is doing what you think it is doing. Testing both small chunks of code and entire functions will help you isolate bugs or other mistakes. The specific tests you use will depend on the goals and functions of your package. As an example, if you are calculating the value of a function and its first and second derivatives, you can use the method of finite differences to check that the derivatives are in the right neighborhood.

1.2 Version Control

We highly recommend you use version control while you work. This will help you track your changes and revert to previous versions of your project. Some methods of version control also make collaboration smoother. Our preferred method of version control is git. Free public Github accounts are available for your remote repository.

2 Creating the Package

Write your R function(s). When your function(s) are ready, use the command `package.skeleton` to create the folders and files for your package. The first argument is the name of your package. The second argument is a vector containing the names of all the R functions you have prepared. Each name should be in quotation marks. To create the `wisdom` package with the `mPower` function, we type

```
package.skeleton( name = "wisdom" , list = c("mPower"))
```

If we had created a second function `morefun`, we would have typed

```
package.skeleton( name = "wisdom" , list = c("mPower", "morefun"))
```

Running the above `package.skeleton` command creates a folder named `wisdom`. In other words, the folder will match the package name. Within this folder are two folders (`R` and `man`) and two files (`DESCRIPTION` and `NAMESPACE`). We describe these files and folders below.

If you inspect the `wisdom` package available on Github, you will see an additional folder (`tests`), which we added ourselves after running `package.skeleton`. Section 2.5 covers the `tests` folder.

2.1 Description

The description file created by `package.skeleton` looks like this:

```
Package: YourPackage
Type: Package
Title: What the package does (short line)
```

```
Version: 1.0
Date: 2017-10-13
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What license is it under?
```

It is your job to open this file and complete it. In particular, you need to add the title, the author (you!), the maintainer and the maintainer's email address (probably also you), a description of the package, and the license you have chosen for your package. You can find a little bit of information on licenses in section 4.3 and a lot of information on the Writing R Extensions site.

We also recommend adding the following line in the description file:

```
ByteCompile: TRUE
```

This compiles your code so that it can run faster each time the function is called. This makes package installation take slightly longer. However, a package only needs to be installed once (unless the package is updated) while functions are often used many times, so the compilation is probably worth the upfront cost.

If/when you update your package, be sure to edit the version number and date in the description file.

For a basic package, this is all you need to know about the description file. If you are more advanced (e.g. including vignettes), you can find more details on the namespace file at Writing R Extensions.

2.2 Namespace

In your namespace file, use `export` and list the function(s) in the package that you want to make available to users. This is necessary to make your function(s) available to users. The `wisdom` package has the most basic namespace possible, as it exports a single function, as shown below.

```
export(mPower)
```

If your package relies on any other packages, you must `import` that package by listing it in the namespace. In the example namespace below, the function depends on R packages `glmm` and `trust` and so it imports them.

```
export(mPower)

import(glmm)
import(trust)
```

For a basic package, this is all you need to know about namespaces. If you are more advanced (e.g. writing S3 or S4 methods, calling C from R), you can find more details on the namespace file at Writing R Extensions.

2.3 R Files

The `R` folder created by `package.skeleton` will contain a .R file for each function you listed in `package.skeleton`. If you wish to edit your package's R code, edit these files. If you wish to add additional R functions after running `package.skeleton`, you certainly can: simply add the function in a .R file to this folder, add the corresponding documentation in the `man` folder, and edit the namespace file to export the function.

2.4 Documentation

The `man` folder created by `package.skeleton` will contain a .Rd file for each function and for the package as a whole. The purpose of these files is to store the information that will help users learn how to use your function(s). This is the documentation that pops up when a user types `?mPower` in the R console. Each function that you export (i.e. make available to users) must be documented. You can delete .Rd files for functions you will not export (e.g. helper functions).

It is your job to open and complete each of these .Rd files. This process is fairly straightforward. You can look at other packages' documentation if you are uncertain (although some packages have terrible documentation!).

If you add an R function to your package after running `package.skeleton`, make sure to add the corresponding .Rd file.

2.5 Tests

Ensuring that your package does what you think it does is essential. We highly recommend creating a `tests` folder. In this folder, create .R files that test your code. You have written your functions generally, but you can test the functions by comparing them against specific examples. When you check your package (`R CMD check`), the tests will run and the results will be `R CMD check` output and log file.

The `all.equal` function is useful for ensuring two numbers (or vectors) are equal. The test file in `wisdom` demonstrates the use of `all.equal`. To check that the `mPower` function is calculating powers correctly, we choose a few specific examples and check the `mPower` results against the results we calculate ourselves.

When you write these tests, try to think creatively. It is easy to imagine testing the square of a positive number. What other values of x and m might someone enter? They can enter any real x, so we test $x < 0$. Additionally, the easiest power to imagine is integer m, but we do not require that m is an integer. The test in the `wisdom` package also checks the function for a few values of m between 0 and 1.

3 Building, Checking, and Installing the Package

When you are satisfied with your code, documentation, description file, and namespace file, it is time to build and check the R package. The first step is to create a "tarball," a compressed version of your package. To do this, navigate to the parent directory of your R package and type the following:

```
R CMD build wisdom
```

The time needed to build your package will increase as you add functions and tests to your package. The tarball produced will contain the version number listed in your package's description file. The `wisdom` package is version 1.0, as the description file shows, so the tarball name includes 1.0.

Next, you will check your package by checking the tarball you have created. Type

```
R CMD check wisdom_1.0.tar.gz
```

to check your package. Once your package passes `R CMD check`, you can install it by typing

```
R CMD install wisdom_1.0.tar.gz
```

4 Additional Considerations

4.1 Computational Stability

Before programming, it is wise to look at each equation in your design document and consider the computational stability. We cover a few items that we have used in our own R package production, but this list is by no means comprehensive.

4.1.1 Big Numbers and Close-to-Zero Numbers

If you are working with large numbers or numbers that are close to zero, you will need to take care because computers can keep track of only so many digits. Once a number is large enough, computers cannot distinguish it from ∞. Similarly, once a number gets close enough to zero, computers cannot distinguish it from zero. A solution to this problem is to work with the log of a number rather than the number itself.

4.1.2 Log of Close-to-One Numbers

As you probably know, the log of 1 is 0. If you choose a number close enough to 1 and try to take the log using `log`, R will output 0. If you prefer a more accurate estimation of the log, use the function `log1p`, which uses a Taylor series expansion around 1.

4.1.3 Catastrophic Cancellation

Consider the variance calculations

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

and

$$\left[\frac{1}{n}\sum_{i=1}^{n} x_i^2\right] - [\bar{x}]^2 .$$

If we consider the equations in a purely mathematical sense, we know these two calculations will yield identical results. However, when we program the variance calculation, the former is more stable due to the order of the summation and subtraction.

In the latter calculation we sum n items for one summation and n items for the other. Recall R can keep track of a limited number of digits for each of these sums. If the data set is large, each of these sums could be quite large. If variance is low, the first sum may be just barely larger than the second. If we have a large data set with low variability, we will lose most of our digits when we subtract one sum from the other. Suddenly losing many digits is called "catastrophic cancellation."

In the former calculation, we first calculate the mean and then we take n differences, square them, and sum them. If the data set is large but the variance is low, we will be taking many differences and many of these differences will be small. When we square them and sum them, we add each term's contribution one at a time. That is, the sum will inch up to produce the variance, rather than crash suddenly due to catastrophic cancellation.

4.2 Submitting to CRAN

If you intend to submit your package to CRAN, you will need to alter your check slightly to meet CRAN's specifications. Build your tarball as usual and check it with the following:

```
R CMD check wisdom.1.0.tar.gz --as-cran
```

When your package check returns no errors, check your package on other platforms. For example, you can use WinBuilder to check your package on the Windows platform. Once you are confident your package produces no errors, you may upload the tarball to CRAN. Do not upload a package that cannot pass `R CMD check`!

4.3 Licenses

You may choose from a variety of licenses. The Writing R Extensions site provides information to help you select an appropriate license.

On one end of the spectrum is the MIT license. If you select this license, anyone can take any/all of your code to produce software, which they can then sell. The `aster` package uses this license.

Another license, the GPL-2 license, allows others to copy your code but requires any copy-cats to provide users with the same rights and freedoms the copy-cat received from your work. That is, the copy-cat cannot choose a license that is more restrictive than your own. The `glmm` package uses this license. The author selected the license because she wanted others to be able to use her code and ideas, but did not want others to profit off it.

4.4 Vignettes

Every R package produces a manual, but the helpfulness of the manual is limited due to the layout of the manual. Functions are listed in alphabetical order rather than the order a user might use them in. A vignette is a document that walks the user through one or more examples to demonstrate how to use the package. An example of a vignette can be found on `cknudson.com` or at `https://cran.r-project.org/web/packages/glmm/vignettes/intro.pdf`.

4.5 Another Note on Tests

If your package uses random numbers (e.g. Monte Carlo), you must make sure you use the same random numbers in your function's calculation and in your check. In your R code, you can create an option to keep track of the random numbers used in your function. You can then use those same random numbers when you write a test for your function. See R package `glmm`'s glmm.R function as an example of optionally outputting the random numbers used. (The random numbers are referred to as `u.star`). The section `objfunNOC` in R package `glmm`'s test file objfunTest.R demonstrates using the stored random numbers `u.star`) to check the package's original R function.

If your package calculates derivatives, you can approximate the derivative using finite differences. If the finite difference is not similar to the derivative calculated by your package, you know that either your function or its derivative is calculated incorrectly.

If your package calls C, C++, or Fortran from R, you should check that the C/C++/Fortran code is producing what you think it should be producing. First code the function in R, since R is much easier than C/C++/Fortran. If the R output does not match the C output, you know you have a problem.

4.6 Useful Links

Writing R Extensions: `https://cran.r-project.org/doc/manuals/r-release/R-exts.html`

PDF version of guide: `https://github.com/knudson1/WSDS2017/tree/master/guide`

R package `wisdom`: `https://github.com/knudson1/WSDS2017/tree/master/wisdom`