In Sum

On many campuses political (and other social) scientists doing field research are faced with educating IRB members and administrative staff about the ways in which their methods differ from the experimental studies performed in hospitals and laboratories. Understanding the federal regulations can put researchers on more solid footing in pointing to permitted research practices that their local Boards may not recognize. And knowing IRB-speak can enable clearer communications between researchers and Board members and staff. Though challenging, educating staff as well as Board members potentially benefits all field researchers, graduate students in particular, some of whom have given up on field research due to IRB delays, often greater for research that does not fit the experimental model (van den Hoonard 2011).

IRB review is no guarantee that the ethical issues relevant to a particular research project will be raised. Indeed, one of our concerns is the extent to which IRB administrative processes are replacing research ethics conversations that might otherwise (and, in our view, should) be part of departmental curricula, research colloquia, and discussions with supervisors and colleagues. Moreover, significant ethical matters of particular concern to political science research are simply beyond the bounds of US IRB policy, including recognition of the ways in which current policy makes "studying up" (i.e., studying societal elites and other power holders) more difficult.

Change may still be possible. In July 2011, OHRP issued an Advanced Notice of Proposed Rulemaking, calling for comments on its proposed regulatory revisions. As of this writing, the Office has not yet announced an actual policy change (which would require its own comment period). OHRP has proposed revising several of the requirements discussed in this essay, including allowing researchers themselves to determine whether their research is "excused" (their suggested replacement for "exempt"). Because of IRB policies' impact, we call on political scientists to monitor this matter. Although much attention has, rightly, been focused on Congressional efforts to curtail National Science Foundation funding, as IRB policy affects *all* research engaging human participants, it deserves as much disciplinary attention.

References

Schrag, Zachary M. 2010. *Ethical Imperialism: Institutional Review Boards and the Social Sciences, 1965-2009.* Baltimore, MD: Johns Hopkins University Press.

Schwartz-Shea, Peregrine and Yanow, Dvora. 2014. Field Research and US Institutional Review Board Policy. Betty Glad Memorial Symposium, University of Utah (March 20-21). `http://poli-sci.utah.edu/2014-research-symposium.php`.

Stark, Laura. 2012. *Behind Closed Doors: IRBs and the Making of Ethical Research.* Chicago: University of Chicago Press.

US Code of Federal Regulations. 2009. Title 45, Public Welfare, Department of Health and Human Services, Part 46, Protection of human subjects. `http://www.hhs.gov/ohrp/humansubjects/guidance/45cfr46.html`.

van den Hoonaard, Will. C. 2011. *The Seduction of Ethics.* Toronto: University of Toronto Press.

Yanow, Dvora and Schwartz-Shea, Peregrine. 2008. "Reforming institutional review board policy." *PS: Political Science & Politics* 41(3): 484-94.

Andreoni, James. 1989. "Giving with Impure Altruism: Applications to Charity and Ricardian Equivalence." *The Journal of Political Economy* 97(6): 1447-58.

Building and Maintaining R Packages with `devtools` and `roxygen2`

Jacob Montgomery
Washington University in St. Louis
jacob.montgomery@wustl.edu

Ryan T. Moore
American University
rtm@american.edu

Political methodologists increasingly develop complex computer code for data processing, statistical analysis, and data visualization – code that is intended for eventual distribution to collaborators and readers, and for storage in replication archives.[1] This code can involve multiple functions stored in many files, which can be difficult for others to read, use, or modify. In many cases, even loading the various files containing the needed functions and datasets can be a time-consuming chore.

For researchers working in R (R Core Team 2014), creating a package is an attractive option for organizing and distributing complex code. A basic R package consists of a set of functions, documentation, and some metadata. Other components, such as datasets, demo, or compiled code may

[1] Supplementary materials, including the code needed to build our example R package, are available at `https://github.com/jmontgomery/squaresPack`.

also be included. Turning all of this into a formal R package makes it very easy to distribute it to other scholars either via the Comprehensive R Archiving Network (CRAN) or simply as a compressed folder. Package creation imposes standards that encourage both the proper organization and coherent documentation of functions and datasets. Packages also allow users to quickly load and use all the relevant files on their systems. Once installed, a package's functions, documentation, datasets, and demonstrations can be accessed using R commands such as `library()`, `help()`, `data()`, and `demo()`.

However, transforming R code into a package can be a difficult and tedious process requiring the generation and organization of files, metadata, and other information in a manner that conforms to R package standards. It can be particularly difficult for users less experienced with R's technical underpinnings. In this article, we demonstrate how to develop a simple R package involving only R code, its documentation, and the necessary metadata. In particular, we discuss two packages designed to streamline the package development process – `devtools` and `roxygen2` (Wickham 2013; Wickham, Danenberg, and Eugster 2011). We begin by describing the basic structure of an R package and alternative approaches to package development, maintenance, and distribution. We compare the steps required to manually manage files, directories, and metadata to a more streamlined process employing the `devtools` package. We conclude with a discussion of more advanced issues such as the inclusion of datasets and demo files.[2]

1. R Package Basics

R package development requires building a directory of files that include the R code, documentation, and two specific files containing required metadata.[3] In this section, we walk through the basic components of an R package. As a running example, we create an R package containing two functions, which are stored in separate files named `addSquares.R` and `subtractSquares.R`.

```
## Function 1: Sum of squares
addSquares <- function(x, y){
  return(list(square=(x^2 + y^2), x = x, y = y))
  }

## Function 2: Difference of squares
subtractSquares <- function(x, y){
  return(list(square=(x^2 - y^2), x = x, y = y))
  }
```

We build the package in the directory `~/Desktop/MyPackage/`, where a `*.R` file containing each function is stored. The R package resides in a single directory whose title matches the package name. The directory must contain two metadata files, called `DESCRIPTION` and `NAMESPACE`, and two subdirectories containing the relevant R code and documentation. Thus, our `squaresPack` package will be structured as follows.

```
squaresPack
├── DESCRIPTION
├── NAMESPACE
├── R
│   ├── addSquares.R
│   └── subtractSquares.R
└── man
    ├── addSquares.Rd
    └── subtractSquares.Rd
```

Populating the package directory consists of four basic steps. First, we store all R source code in the subdirectory `R`. A good standard is to include each R function as a separate file. Second, corresponding documentation should accompany all functions that users can call. This documentation, which explains the purpose of the function, its inputs, and the values of any output, is stored in the subdirectory `man`. For example, the file `addSquares.Rd` would appear as follows.

```
\name{addSquares}
\alias{addSquares}
\title{Adding squared values}
\usage{
 addSquares(x, y)
}
\arguments{
 \item{x}{A numeric object.}
 \item{y}{A numeric object with the same dimensionality as \code{x}.}
}
\value{
 A list with the elements
 \item{squares}{The sum of the squared values.}
 \item{x}{The first object input.}
 \item{y}{The second object input.}
}
\description{
 Finds the squared sum of numbers.
}
\note{
 This is a very simple function.
}
\examples{
myX <- c(20, 3); myY <- c(-2, 4.1)
addSquares(myX, myY)
}
\author{
  Jacob M. Montgomery
}
```

Third, the directory must contain a file named `DESCRIPTION` that documents the directory in a specific way. The `DESCRIPTION` file contains basic information including the package name, the formal title, the current version number, the date for the version release, and the name of the author and maintainer. Here we also specify any dependencies on other R packages and list the files in the `R` subdirectory.

```
Package: squaresPack
Title: Adding and subtracting squared values
Version: 0.1
```

[2]The code and examples below were written for Mac OS X (10.7 and 10.8) running `R` version 3.1.0 with `devtools` version 1.5. Some adjustment may be necessary for authors using Linux. R package creation using Windows machines is not recommended. A useful online tutorial for creating an R package in RStudio using `devtools` and `roxygen2` is currently available at: `https://www.youtube.com/watch?v=9PyQlbAEujY`

[3]The canonical source on package development for R is "Writing R Extensions" (R Core Team 2013).

```
Author: Jacob M. Montgomery and Ryan T. Moore
Maintainer: Ryan T. Moore <rtm@american.edu>
Description: Find sum and difference of squared values
Depends: R (>= 3.0.0)
License: GPL (> = 2)
Collate:
  'addSquares.R'
  'subtractSquares.R'
```

Finally, the `NAMESPACE` file is a list of commands that are run by R when the package is loaded to make the R functions, classes, and methods defined in the package "visible" to R and the user. As we discuss briefly below, details on class structures and methods can be declared here. For `squaresPack`, the `NAMESPACE` file tells `R` to allow the user to call our two functions.

```
export(addSquares)
export(subtractSquares)
```

2. Approaches to Package Development and Maintenance

Authors can create and update packages in several ways, arrayed on a continuum from "very manual" to "nearly automated." At the "very manual" end, the author starts by creating the directory structure, each of the required metadata files, and a documentation file for each function. A "semi-manual" approach initializes the package automatically, but then requires that maintainers update the metadata and create documentation files for new functions as they are added. We describe this latter approach to build readers' intuition for what happens behind the scenes in the "nearly automated" approach we detail in Section 2.2, and because this approach requires nothing beyond base R.

2.1. Semi-manual Package Maintenance

A "semi-manual" procedure automatically initializes the package, but may require substantial bookkeeping as development proceeds.

Package creation: After the author loads the required functions into her workspace, she provides `package.skeleton()` with the package name and a list of the functions to be included.

```
setwd("~/Desktop/MyPackage/") ## Set the working directory
source("addSquares.R") ## Load functions into workspace
source("subtractSquares.R")
package.skeleton(name = "squaresPack",
                 list = c("addSquares", "subtractSquares"))
```

This creates the package directory using the proper structure, generates blank documentation files with the appropriate file names, and includes a helpful 'Read-and-delete-me' file that describes a few of the next steps. After the package is created, the author edits the `DESCRIPTION`, `NAMESPACE`, and help files, and the package is ready to compile and submit to CRAN. To compile the package, check it for errors, and install it on the author's instance requires three steps (shown below for the Terminal prompt in the Mac OS),

```
R CMD build --resave-data=no squaresPack
R CMD check squaresPack
R CMD INSTALL squaresPack
```

Package maintenance and submission: Superficially, the process described above may not seem cumbersome. However, calling `package.skeleton()` again (after deciding to add a new function, for example) will overwrite the previously-created directory, so any changes to documentation or metadata will be lost. Thus, after the original call to `package.skeleton()`, the author should manually add new data, functions, methods, and metadata into the initial skeleton. Adding new arguments to an existing function requires editing associated help files separately. Thus, a minimal list of required steps for updating and distributing an R package via this method includes the steps shown below.[4]

1. Edit `DESCRIPTION` file
2. Change R code and/or data files.
3. Edit `NAMESPACE` file
4. Update `man` files
5. `R CMD build --resave-data=no pkg`
6. `R CMD check pkg`
7. `R CMD INSTALL pkg`
8. Build Windows version to ensure compliance by submitting to: `http://win-builder.r-project.org/`
9. Upload (via Terminal below, or use other FTP client):
   `> ftp cran.r-project.org`
   `> cd incoming`
   `> put pkg_0.1-1.tar.gz`
10. Email R-core team: `cran@r-project.org`

This approach comes with significant drawbacks. Most importantly, editing the package requires altering multiple files stored across subdirectories. If a new function is added, for instance, this requires updating the `R` subdirectory, the `DESCRIPTION` file and usually the `NAMESPACE` file. In more complicated programming tasks that involve class structures and the like, such bookkeeping tasks can become a significant burden. Moreover, the process of actually building, checking, and submitting a package can involve moving between multiple user directories, user interfaces, and software.

We have authored four R packages over the course of the last six years. To organize the manual updating steps, one of us created an 17-point checklist outlining the actions

[4] We omit some maintenance details such as updating the LICENSE file, the Changelog, and unit testing.

required each time a package is edited. We expect that most authors will welcome some automation. The packages `devtools` and `roxygen2` can simplify package maintenance and allow authors to focus more on improving the functionality and documentation of their package.

2.2. `devtools` and `roxygen2`

`devtools` streamlines several steps: it creates and updates appropriate documentation files, it eliminates the need to leave R to build and check the package from the terminal prompt, and it submits the package to `win-builder` and CRAN and emails the R-core team from within R itself. After the initial directory structure is created, the only files that are edited directly by the author are contained in the `R` directory (with one exception – the `DESCRIPTION` file should be reviewed before the package is released). This is possible because `devtools` automates the writing of the help files, the `NAMESPACE` file, and updating of the `DESCRIPTION` file relying on information placed directly in `*.R` files.

There are several advantages to developing code with `devtools`, but the main benefit is improved workflow. For instance, adding a new function using more manual methods requires creating the code in a `*.R` file stored in the `R` subdirectory, specifying the attendant documentation as a `*.Rd` file in the `man` subdirectory, and updating the `DESCRIPTION` and `NAMESPACE` files. In contrast, developing new functions with `devtools` requires only editing a single `*.R` file, wherein the function and its documentation are written simultaneously. `devtools` then updates the documentation (using the `roxygen2` package), and package metadata with no further attention.

Writing `*.R` files: Thus, one key advantage of using `devtools` is that the `*.R` files will themselves contain the information for generating help files and updating metadata files. Each function is accompanied by detailed comments that are parsed and used to update the other files. Below we show how to format the `addSquares.R` file to create the same help files and `NAMESPACE` file shown above.

```
#' Adding squared values
#'
#' Finds the sum of squared numbers.
#'
#' @param x A numeric object.
#' @param y A numeric object with the same dimensionality as \code{x
    }.
#'
#' @return A list with the elements
#' \item{squares}{The sum of the squared values.}
#' \item{x}{The first object input.}
#' \item{y}{The second object input.}
#' @author Jacob M. Montgomery
#' @note This is a very simple function.
#' @examples
#'
#' myX <- c(20, 3)
#' myY <- c(-2, 4.1)
#' addSquares(myX, myY)
#' @rdname addSquares
#' @export
addSquares<- function(x, y){
  return(list(square=(x^2 + y^2), x = x, y = y))
}
```

The text following the `#'` symbols is processed by R during package creation to make the `*.Rd` and `NAMESPACE` files. The `@param`, `@return`, `@author`, `@note`, `@examples`, and `@seealso` commands specify the corresponding blocks in the help file. The `@rdname` block overrides the default setting to specify the name of the associated help file, and `@export` instructs R to add the necessary commands to the `NAMESPACE` file. We now walk through the steps required to initialize and maintain a package with `devtools`.

Setting up the package: Creating an R package from these augmented `*.R` files is straightforward. First, we must create the basic directory structure using

```
setwd("~/Desktop/MyPackage/") ## Set the working directory
create("squaresPack")
```

Second, we edit the `DESCRIPTION` file to make sure it contains the correct version, package name, etc. The `create()` call produces a template file. The author will need to add some information to this template `DESCRIPTION` file,[5] such as

```
Author: Me
Maintainer: Me <me@myemail.edu>
```

`devtools` will automatically collate all R files contained in the various subdirectories. Third, place the relevant R scripts in the `R` directory. Finally, making sure that the working directory is correctly set, we can create and document the package using three commands:

```
current.code <- as.package("squaresPack")
load_all(current.code)
document(current.code)
```

The `as.package()` call loads the package and creates an object representation of the entire package in the user's workspace. The `load_all()` call loads all of the R files from the package into the user's workspace as if the package was already installed.[6] The `document()` command creates the required documentation files for each function and the package, as well as updates the `NAMESPACE` and `DESCRIPTION` files.

Sharing the package: Next, the author prepares the package for wider release from within R. To build the package, the author runs `build(current.code, path=getwd())`. The analogous `build_win()` command will upload the package to the `http://win-builder.r-project.org/` website. This builds the package in a Windows environment and emails the address of the maintainer in the `DESCRIPTION` file with results in about thirty minutes. Both of these compressed files can be uploaded onto websites, sent by email,

[5] The `DESCRIPTION` file should not contain any blank lines. If the template file contains any, these will either need to be deleted or filled in.

[6] The help files and demo files will only be available using after running `install(current.code)`, which is equivalent to `R CMD INSTALL` in the Terminal.

or stored in replication archives. Other users can download the package and install it locally.

The package can be submitted to CRAN without the need to leave R. We provide a minimal checklist for editing and submitting an existing R package using `devtools`:

1. Edit R code and/or data files
2. Run `as.package()`, `load_all()`, and `document()`
3. Check the code: `check(current.code)`
4. Make a Windows build: `build_win(current.code)`
5. Double-check the `DESCRIPTION` file
6. Submit the package to CRAN: `release(current.code, check=FALSE)`

The `check()` command is analogous to the `R CMD check` from the terminal, but it also (re)builds the package. Assuming that the package passes all of the required checks, it is ready for submission to CRAN. As a final precaution, we recommend taking a moment to visually inspect the `DESCRIPTION` file to ensure that it contains the correct email address for the maintainer and the correct release version. Finally, the `release()` command will submit the package via FTP and generate the required email. This email should come from the same address listed for the package maintainer in the `DESCRIPTION` file.

3. Extensions: Documentation, Data, Demos, and S4

Often, R packages include additional documentation, datasets, and commands that can be executed using the `demo()` function. These can illustrate the package's functionalities, replicate results from a published article, or illustrate a set of results for a collaborator. Some authors may also work in the S4 framework, which requires more documentation and some tricks for setting up the package to pass CRAN checks. A somewhat more developed R package might consist of a directory structured as follows.

```
squaresPack
├── DESCRIPTION
├── NAMESPACE
├── R
│   ├── addSquares.R
│   ├── subtractSquares.R
│   ├── exampleDataset.R
│   └── squaresPack-package.R
├── man
│   ├── squaresPack.Rd
│   ├── addSquares.Rd
│   ├── subtractSquares.Rd
│   └── exampleDataset.Rd
└── data
    └── exampleDataset.rda
└── demo
    ├── 00Index
    ├── addSquares.R
    └── subtractSquares.R
```

3.1. Package Documentation

One common feature of many packages is some simple documentation of the package itself. Using `devtools`, this requires the author to include a chunk of code in some file in the `R` subdirectory. In our example, we include a file called `squaresPack-package.R` containing the following code. (Note the use of the `@docType` designation and that no actual R code is associated with this documentation.)

```
#' squaresPack
#'
#' The squaresPack package performs simple arithmetic calculations.
#' @name squaresPack
#' @docType package
#' @author Ryan T. Moore: \email{rtm@@american.edu} and
#' Jacob M. Montgomery: \email{jacob.montgomery@@wustl.edu}
#' @examples
#'
#' \dontrun{
#' demo(addSquares)
#' demo(subtractSquares)
#' }
#'
NULL
```

3.2. Datasets

Another common feature of many `R` packages is the inclusion of datasets. Datasets are typically stored as `*.rda` objects and must be located in the `data` subdirectory. Documenting the dataset is similar to documenting the package itself. In our example, we created a separate `exampleDataset.R` file with the following content.

```
#' Example Dataset
#'
#' This line could include a brief description of the data
#'
#' The variables included in the dataset are:
#' \itemize{
#' \item\code{Variable1} A vector of random numbers
#' \item\code{Variable2} Another vector of random numbers
#' }
#'
#' @name exampleDataset
#' @docType data
NULL
```

3.3. Demo Files

The demo file provides examples for particular functions or the package as a whole. Demo files should contain a single R script that will be run when the user calls `demo(addSquares)` or `demo(subtractSquares)`. Since this command will also be run during the normal R check, authors may want to omit any extremely slow or time-consuming command.

As the directory structure above shows, the `demo` subdirectory must include an index file named `00Index` listing the included demo files, one per line, with a short description:

```
addSquares      Demo file for addSquares
subtractSquares Demo file for subtractSquares
```

Each demo file is a `*.R` file that ends with code that is run when the user types, for example, `demo(addSquares)`:

```
dx <- 1:2
dy <- 3:4
addSquares(dx, dy)
```

3.4. S4 Considerations

Finally, some authors may work in an S4 environment, which requires the specification of both class structures, generics, and class-specific methods.[7] In S4 development, every class, subclass, and method must have a help file. To handle this, one can include a list of 'aliases' in the help files. That is, one can make one help file for the class definition also work for some of the more trivial class methods that may not require their own documentation. To do this, one includes multiple class-specific methods in the `@alias` block. One can point multiple classes and methods to the same help file using the `@rdname` command.

4. Conclusion

We illustrate how the `devtools` package can aid package authors in package maintenance by automating several steps of the process. The package allows authors to focus on only editing `*.R` files since both documentation and metadata files are updated automatically. The package also automates several steps such as submission to CRAN via ftp.

While we believe that the `devtools` approach to creating and managing R packages offers several advantages, there are potential drawbacks. We routinely use other of Hadley Wickham's excellent packages, such as `reshape`, `plyr`, `lubridate`, and `ggplot2`. On one hand, each of them offers automation that greatly speeds up complex processes such as attractively displaying high-dimensional data. However, it can also take time to learn a new syntax for old tricks. Such frustrations may make package writers hesitant to give up full control from a more manual maintenance system. By making one's R code conform to the requirements of the `devtools` workflow, one may lose some degree of flexibility.

Yet, `devtools` makes it simpler to execute the required steps efficiently. It promises to smoothly integrate package development and checks, to cut out the need to switch between R and the command line, and to greatly reduce the number of files and directories that must be manually edited. Moreover, the latest release of the package contains many further refinements, such as building packages directly from GitHub repositories, creating vignettes, and creating "clean" environments for code development. While developing R packages in a manner consistent with `devtools` requires re-learning some techniques, we believe that it comes with significant advantages for speeding up development and reducing the frustration commonly associated with transforming a batch of code into a package.

References

Chambers, John M. *Software for Data Analysis: Programming with R*. New York, NY: Springer.

R Core Team. 2013. "Writing R Extensions." `http://cran.r-project.org/doc/manuals/R-exts.html`, Version 3.0.0.

R Core Team. 2014. *R: Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. ISBN 3-900051-07-0.

Wickham, Hadley. 2013. *devtools: Tools to Make Developing R Code Easier*. R package version 1.2. `http://CRAN.R-project.org/package=devtools`.

Wickham, Hadley, Peter Danenberg, and Manuel Eugster. 2011. *roxygen2: In-source documentation for R*. R package version 2.2.2. `http://CRAN.R-project.org/package=roxygen2`.

[7] For additional details on S4 programming, see Chambers (2008).