RStudio, Project Organization, and Getting Data into R

Topics

1. R Studio IDE
2. Organizing projects
3. Getting data into R

RStudio: Interactive Development Environment (IDE) of choice.

You are already familiar with using R in its most basic form ... the command line and a separate script window that comes with Rgui. You don't *need* anything else to use R. However, using an "IDE" can speed up and make more pleasant your coding experience. An IDE just adds a bunch of bells and whistles to your interaction with R. In the past I haven't recommended any particular IDE because I felt that each of them had problems and there were cross-platform issues with all of them. That, however, was before RStudio. The developers of this free, open source product got it right and this is, by far, the best IDE out there. It works flawlessly on any platform and has all the *right* bells and whistles. Because I think it will make your life so much easier, we are going to spend the first few minutes of class learning how to use it. If you are using your own laptop, download and install RStudio (Desktop) and then start the program. If you are using the lab computers: Start→All Programs→RStudio.

→ Go to the Using RStudio Documentation, and read (and try code snippets), in order (there are others you can also check out, but be sure to do these ones):

1. Working in the Console
2. Editing and Executing Code
3. Code Folding and Sections
4. Navigating Code

Project Organization

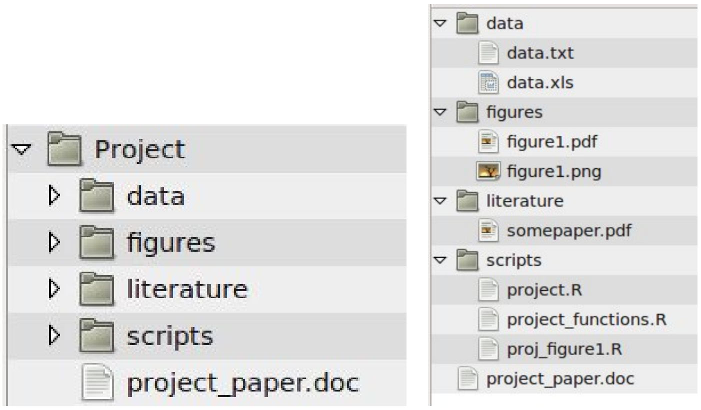
It is useful to figure out how to store data, analyses, scripts, figures, and text in a way that facilitates future work. There are a number of benefits to a good organizational system:

- It makes it easy for you to pick up where you left off the next time you return to a project
- It makes it trivial to collaborate because you can send a single folder to your colleague(s) for modification
- It makes it possible for others (not just your colleagues) to exactly repeat your analyses in the near or distant future
- It saves you tons of time that would otherwise be spent plowing through folders and files and trying to remember where things are and what you did last.

In my opinion, good project organization requires, at a minimum, three things:

1. **Standard folder and file heirarchy.** There are many different ways to do this. I've settled on what I think is a nice balance between having enough folders to organize a project but not so many that most of your time is spent navigating your folder heirarchy (Fig. 1). Every project (a manuscript, for example) is housed in a folder (named "Project" in the example, Fig. 1(a)). The `Project` folder has 4 folders (rarely a few more). The `data` folder houses my data, usually in text and/or spreadsheet format (see comments about data below). Obviously, the figures are written to the `figures` folder by scripts which reside in the `scripts` folder. The text resides in the project folder itself and, if you are using LaTeX[1], is automatically

[1]A topic for another course, LaTeX is a fantastic way to write documents (for example, all the handouts for this course).



(a) Overall project organization (b) Expanded view with example files

Figure 1: A functional project organization scheme.

updated any time data, scripts, or figures change. (Sometimes I also temporarily house project-specific literature in its own folder within the project folder.) The scripts folder deserves a bit more attention so is discussed in detail below.

2. **Foolproof data storage.** Your data are sacrosanct. It is far too easy to accidentally alter or destroy any or all of your data, so it is critical that you have good strategies for preventing data corruption. I generally have my data in two forms: a spreadsheet where it is easy to enter data to begin with, and a text file (tab- or comma-delimited) which is easy to load into R, but difficult to edit. The spreadsheet is unnecessary if your data collection allows for direct writing to text files. You can even go so far as to lock or protect these files from editing (I haven't found this necessary). It helps to see your data and your analyses as two entirely separate entities. R scripts allow you to load in and mess with your data in any number of ways ***without altering the original data***. I can't emphasize enough how important and useful this is. On the one hand, it keeps your data safe. On the other, you have complete freedom to do whatever you want with your data, knowing you won't mess it up.

3. **Well-documented R scripts.** In combination with a standard folder heirarchy and secure data storage, good R scripts allow you to repeat any analyses at any time with your current data, modified versions of that data, and even with new datasets. I've already outlined a set of best practices for scripting (see the first session handout), but the example script on page 4 demonstrates how a well-organized script can work with a good folder heirarchy.

   - The first line of the script sets the working directory to the project folder itself. This makes it easy to collaborate on projects because changing only this line to suit a specific system will make everything else work.
   - Next I load any libraries that are necessary. I do this at the beginning of the script for transparency: the functions that you are using from those libraries may be embedded deep in the script and therefore when you look at this analysis again or share it with a colleague, it may not be obvious that the library is required.
   - Next, I read in any project-specific functions (if they exist). Again, this is at the top of the script for transparency (also note that I have to tell R to look in the `scripts` folder because the working directory is the main project folder).
   - I then load in the data, run analyses, and make figures. Note that figures can be written directly to pdf (or png, jpg, tiff, etc.), and that I write them to files in the `figures` folder.

In the example I've given in Fig. 1(b), I have a main project script (`project.R`), a functions script (`project_functions.R`), and even a figure script (`proj_figure1.R`). Why put these things in separate scripts? As with deciding on folder and file heirarchies, you are trying to optimize legibility and functionality. On the one hand, you could put every analysis and/or figure in it's own script, making each individual script easy to read and understand. The problem is that you may end up with so many individual files that you can't remember which is which, nor can you remember their order. This makes it difficult to match analyses up with text and can cause huge problems if one script depends on another. The opposite approach would be to place everything in one script. This takes care of matching order of analyses with text and of dependence of some analyses on others. It also makes it easy to keep track of the files associated with a project as there aren't many! However, your one script may be very hard to read because it'll be so long! I find that, except for the most basic analyses, my main project script can grow very large very quickly. So if you use only one script, you may spend all your time scrolling through code trying to find a particular section. (This is true even for well-documented scripts).

After much experimentation (and frustration), I've settled on a compromise between these two extremes. I have a main project script (`project.R` in the example) that is a "parent" script. In theory, it is the only script that you need to open because all other scripts are called by it. This keeps everything in one place, while still maintaining legibility because bulky, unwieldy bits of code can be parceled out into subscripts so you don't have to look at them unless you want to. If you use this approach, it is important to document in your subscripts what they depend on and where they are called from (script and line number).

Getting data into R manually

There are a number of ways to get data into R. You have already had practice entering data *manually*, either at the command-line or in a script. We'll get more practice and learn some very useful ways to manipulate data with some in-class exercises.

In-class exercises

→ Download Chapter 2, Getting Data into R of A Beginner's Guide to R from the UW libraries. (This link will only work if you are connecting from a campus computer or through a VPN from off campus)

→ Download the zip archive with necessary files (Save the file to your computer after clicking the link, then unzip and you will have a folder with all of the necessary files).

The unzipped folder contains a number of files (see the README file in the archive for a description of contents). First of all, rather than make you go through the drudgery of typing all the commands in as you follow along with the chapter, I have written a script (`BGR_chap2.R`) with all of the chapter commands as well as some additional notes and questions. Another script (`BGR_chap2_answers.R`) has answers to the questions and exercises posed in the first script.

→ Open the BGR_chap2.R script in RStudio and practice sending the commands to the command line as you follow along in the text. Try the exercises in the script (indicated with #?-) and check your answers with the BGR_chap2_answers.R script. Finish through Section 2.1.6.

Homework

1. Finish BGR Chapter 2. As you are doing so, make notes of the things that don't make sense to you.

2. Setup a project heirarchy for one of your own projects. You can use my approach or any approach that makes sense to you. Be prepared to discuss why you set it up as you did.

Example script, part II

```
# project.R
#
#  This script does some neat things by relying on interesting techniques ...
#
#  Depends: data.txt
#           project_functions.R
#
#  created 10 Sep 2015 by M. Dillon
#
#  last modified 11 Sep 2015 by M. Dillon
#
#  More detailed description if necessary--requirements, structure of data,
#    references, etc. Much of this may go in comments throughout the script.
##############################################################################

## set working directory-- all calls are relative to this directory,
##   so someone else wanting to repeat this analysis can simply change
##   this line to point to their project directory, then all the rest should work.
setwd(``somedir/otherdirs/Project'')

## libraries
library(Hmisc)

## non-standard functions
source(``scripts/project_functions.R'')

## load data
data <- read.table(``data/data.txt'')

## write a figure to file
pdf(``figures/figure1.pdf'')

(some plotting commands)

dev.off()

## alternatively, you might use an external script to generate a particularly
##   complicated figure with lots of code:
source(``figure1.R'')
```