Reproducibility of data analysis

Stephanie Palmer & Stefano Allesina & Graham Smith

September 12-14, 2017

Contents

Goals

This tutorial will cover methods for making your code reproducible, both by you and by others. Producing results that are reproducible by others is the very essence of science, and writing code that is reproducible by **you** is just the first step. In particular, we will discuss methods to ensure that you can:

- reproduce your calculations precisely,
- make your code readable,
- track changes to your code,
- make your code freely accessible to the wider world via github.

Along the way, you will get an introduction to stochastic processes and how they are used to model biological variability. By the end of this tutorial, you should know why it is important to save your seeds and merge your branches. You should also know why reproducible coding practices can help you *now*, even if (indeed, particularly if) you are just learning to code.

Installation notes

For this tutorial, relatively up-to-date versions of `R` and `RStudio` are needed. To install `R`, follow instructions at `cran.rstudio.com`. Then install `Rstudio` following the instructions at `goo.gl/a42jYE`. Download the `ggplot2`, `cowplot`, `stats`, and `RMKdiscrete` packages. You should also install the UNIX utility `git` for OSX or Windows.

Introduction to `git`

`git` is a version control system. Version control is a way to record and organize changes to a project that consists of files and directories. Over time, a version control system like `git` builds a repository of the whole history of your project.

When you start working on a new project, you tell `git` what directory will contain the project. Then `git` takes a snapshot of that directory to create the beginning of your repository. If the project is brand new, this may be a snapshot of an empty directory. After you've done some work, you can tell `git` to take a new snapshot, called a **commit**. All snapshots remain available—you can always recover previously committed versions of files.

`git` is especially important for collaborative projects: everybody can simultaneously work on the project, even on the same file. Conflicting changes are reported, and can be managed using side-by-side comparisons. The possibility of **branching** allows you to experiment with changes (e.g. shall we rewrite the introduction of this paper?), and then decide whether to **merge** them into the project.

Why use Version Control?

If you ever collaborate on writing code, then version control is for you. If you ever horribly break your project and need to move quickly back in time two days (or two weeks), then version control is for you. If you ever need to replicate a figure you made three months ago, version control is for you (I *guarantee* you will need to do this at some point). If you ever need to share your code publicly, then version control is for you. Most likely, version control is for you.

Version control is useful for small projects, and is essential for large collaborative projects. It vastly improves the workflow, efficiency and reproducibility. Without it, it is quite easy to lose track of the status of a manuscript (who has the most recent version?), or lose time (I cannot proceed with my changes, because I have to wait for the edits of my collaborators).

Many scientists keep backup versions of the same project over time or append a date/initials to different versions of the same file (e.g. various drafts of the same manuscript). This manual approach quickly becomes unmanageable, with simply too many files and versions to keep track of in a timely and organized manner. Version control allows you to access all previously committed versions of the files and directories of your project. This means that it is quite easy to undo short-term changes: Bad day? Just go back to yesterday's version! You can also access previous stages of the project: "I need to access the manuscript's version and all the analysis files in exactly the state that they where in when I sent the draft for review three months ago." Checking out an entire project at a certain point in time is easy with a version control system but much more difficult with Dropbox or Google Drive.

Version control makes it trivial to host your code publicly (e.g. on Github or Bitbucket) and to share a robust link to your code in any publication.

Version control might look like overkill at first. However, with a little bit of practice you will automatically run a few commands before you start working on a project, and again once you are done working on it. A small price to pay, considering the advantages. Simply put, using version control makes you a more organized and efficient scientist.

> Stefano's testimonial: "Our laboratory adopted version control for all our projects in 2010, and sometimes we wonder how we managed without it."

Help using GitKraken and `git`

GitKraken is a graphical user interface (GUI) on top of the older command-line interface (CLI) `git`. If you've never used `git` before, GitKraken has a good tutorial: https://support.gitkraken.com/start-here/guide

Glossary

- `commit`: *v.* Take a snapshot of the current progress. *n.* A snapshot of the project at a particular point in time
- **commit graph**: TODO
- `stage`: *v.* Add changes to the set that will be committed. (does not commit!)
- `repository` (`repo`): *n.* Entire history of project. Set of all commits on all branches.
- `branch`: *n.* A lineage of commits. A repository can have multiple branches, and committing changes to one will not affect the others. You can switch between branches (see `checkout`). When you switch branches, all your project's files change to the state they were in at the last commit on that branch. *v.* To create a new branch.
- `checkout`: **TODO**

Workflow

TODO: fix numbers 1) Pull any new changes from your collaborators 1) Work on your project 2) When you've done something meaningful, stage your changes 3) Commit your changes, writing a meaningful commit message 4) Repeat! 5) When you're done for the day, push your changes to the remote repository (e.g. on GitHub).

First Steps: Your First Repo

Let's `init` your first repo from scratch! To do so, in GitKraken select `File > Init Repo`. We'll be creating a "Local Only" repository for this simple example. That just means we'll not be putting any files on any website (such as GitHub). The files are only stored on your computer. Browse for [TODO: Place to put files].

Great! Now we have an empty repository. Let's put something in it. Using a text editor of your choice (Notepad, TextEdit, Sublime, even RStudio), create a file called `origin.txt` with the text "An abstract of an Essay on the Origin of Species...". Save it.

Now go back to GitKraken. At the top of the center panel, an entry has been added, `// WIP`. This stands for "Work In Progress" and it indicates you've made changes to your project that have not been committed to the repository. We have two steps to record the changes to the repository: First, we stage the changes. Second, we commit the staged changes.

TODO: Staging and committing

If you use `git` in the wild, you'll find that [TODO: This is everything]

A Deeper Example

For the remainder of the tutorial, we'll be using a more complicated example based in code hosted on GitHub. TODO

Clone a Repo

The previous section covered how to create your own repo from scratch [TODO: common use case: use someone else's code]

[TODO: clone LotsOBugs repo]

Remote repositories

So far, we have been working with local repositories, hosted in only one computer. When collaborating (or when working on the same project from different computers), it is convenient to host the repository on a server, so that the collaborators can see your changes, and add their own.

There are two main options: a) setup your own server (this is beyond the scope of this introduction); b) host your repositories on a website offering this service (for a fee, or for free).

The most popular option for hosting open source projects, where the whole world can see what and when you commit, is GitHub, which will store all your public repositories for free. In fact, as as student you can get private repositories for free! The Student Developer Pack comes with access to lots of other goodies, but private repositories make it a necessity. They're useful in the early stages of your project when you're paranoid about anyone, let alone the entire world, seeing your hasty hacks. But remember to publicize your repository when you publish! (Note: probably still not HIPAA compliant)

After the initial setup (which is specific to the service you use, and thus not treated here), you only need to add two new commands to your `git` workflow: `pull` and `push`. When you want to work on a project that is tracked by a remote repository, you `pull` the most recent version from the server and work on your local copy using the commands illustrated above. When you are done, you `push` your commits to the server so that other users can see them. Your workflow will look like:

Understanding the Code

TODO: Maybe move the below to the README?

Noise in biological systems

Many of the variables that we observe in biological recordings fluctuate, sometimes because we cannot control all the states of the external and internal experimental system, other times because thermal noise makes the state of the biological system we interrogate inherently variable. Examples of fluctuating quantities in biological systems include: the number of a certain type of molecule in a cell; the number of open channels in a cell; the number of electrical action potentials or "spikes" emitted by a neuron in response to a stimulus; the number of individuals in a population at a particular moment in time; the number of bacterial colonies on a plate. These are all quantities that we can make precise claims about, on average, but cannot specify with certainty for any particular experimental observation.

It is useful to model not only a mean value for a fluctuating variable, but the full shape of its distribution of values. For example, if we observe the firing of neurons in the brain to repeats of the same external stimulus, the precise times of spikes will vary between repeats. By fitting the statistics of this noise to models, we deepen our mechanistic understanding of the neural response. We can test whether or not the "noise" we observe is consistent with a truly random source of output variation, or if it has some structure that tells us about interactions between the neurons and their environment.

Often, noise in biological systems is modeled by what's called a Poisson process, whose values follow a Poisson distribution. The program you wrote to sample the bent coin lottery generated tickets whose statistics follow the binomial distribution. The binomial distribution approches the familiar Poisson distribution, in the limit of a large number of trials, n, or a small probability of the event, p, per trial:

$P_n(k)_{n\to\infty} = \frac{\lambda^k}{k!}e^{-\lambda}$

where λ is the average rate of occurrence of our event in n trials. We have just written down the Poisson distribution. You will see this used as a model for biological variability again and again, either explicitly or implicitly. It is important to think about whether or not it is a good model for the system under study each time you come across it or are deciding to use it for your own research. Notes on deriving the relationships between some common distributions are provided in the readings folder.

Arthropod dispersion

In 1941 and 1942, zoologist LaMont Cole, then working at the University of Chicago, set out to survey species diversity and distribution in the woods and pastures of Kendall County, Illinois. He was particularly interested in which species co-occurred in woods versus grazing land and how their numbers varied with changes in humidity and temperature throughout the year. He laid thick oak boards in a variety of locations and counted the number of "cryptozoic" (animals found under stones, rotten logs, tree bark, etc.) individuals found under the boards several times a week, over the course of a year. For his spatial distribution studies, he aimed to determine whether arthropods distributed themselves randomly or if they had a more complex interaction pattern with each other or with their environment.

Branching and merging [TODO: rewrite]

Most VCSs allow you to *branch* from the main development of the project (i.e., experiment freely with your project without messing up the original version). It is like saying "Save as" with a different name, but with the possibility of easily merging the two parallel versions of the file. Typical examples of a branching point in scientific projects are: a) you want to try a different angle for the Introduction of your manuscript, but you are not sure it's going to be better; b) you want to try to rewrite a piece of code, to see whether it will be faster, but surely you do not want to ruin the current version that is working just fine; c) you want to keep working on the figures, while your collaborator is editing the manuscript. In these cases, you are working on an "experimental feature", while leaving the main project unaltered (or while other people are working on it). Once you are satisfied with your changes, you would like to *merge* them with the main version of the project.

[TODO: Branch to make changes to repo]

References and readings

Journal articles

All of the data used in this tutorial come from original research papers that are in the `readings` folder. Also in the `readings` folder, you will find an article by Roger Peng arguing for setting standards for reproducible research in computational science. It's a short article that we hope you will read and adopt as best practices for your own work.

Books and tutorials

There are very many good books and tutorials on `git`. We are particularly fond of *Pro Git*, by Scott Chacon and Ben Straub. You can either buy a physical copy of the book, or read it online for free.

Both GitHub and Atlassian (managing Bitbucket) have their own tutorials.

A great way to try out Git in 15 minutes is here.

Software Carpentry offers intensive on-site workshops and online tutorials.

GUIs

There are several Graphical User Interfaces for git.

Readability

When you begin programming, you have written just a few programs and functions and it might be possible to hold all of your naming conventions and little tricks in your mind at once. However, wait just a few days – let alone a few months – and you'll have a hard time deciphering, perhaps even running *your own* code if you don't write readable programs. Best practices in modern computational science also dictate that you share your code whenever you publish a result. This means that other people need to be able to read your code; it should run on other computers with other folks sending the input data and specifying or *changing* parameters. This might seem daunting, but a few simple habits will aid in keeping all of your code parsable by other humans, including your own future self.

Commenting your code

The simplest way to make your code more readable is to write comments. Comments begin with # and continue to the end of the line. There are several different types of comments, and each explains what code does, how it's done, or why you did it. For our purposes, we'll divide comments into two broad categories: block comments and in-line comments. Block comments take up the whole of one or more lines and describe meaningful units comprising many lines of code. In-line comments occupy the end of a line of code, or are sandwiched between lines of code, and describe only one or few lines of code.

Block comments

Each meaningful part of your program should have a block comment explaining the purpose of that part (e.g. functions, scripts, the whole program).

‘‘‘

```
# Modeling a Society of Foos
#
# Author:          Barry Allen
# Created:         2017 Sep 15
# Last Modified:   2017 Sep 16
#

make_a_foo <- function() {
  # Constructs a foo.
  #
  # Returns a foo object.

  return(foo)
}

crown_a_foo <- function(foo) {
  # Create a new royal foo from an ordinary foo.
  #
  # Args:
  #   foo - a foo object with no royal status
  # Returns a royal foo
  foo.is_royal = TRUE
  return(foo)
}
```

First, at the top of your file you should include a block comment that describes who wrote the code, when it was last updated, what it does, and how and why someone would use it.

Similarly, the first lines of any function should be comments. These comments should describe what the function does as well as its inputs and outputs. Ideally name and describe each of the function's arguments individually, as well as the return value. For advanced users, there are packages, e.g. `roxygen2`, that will automatically convert such comments into documentation in the `man` folder, that can then be accessed by typing `?` or `help()`.

If you are writing a script, you should include a comment at the top of every meaningful code block. These blocks are somewhat like paragraphs, and like paragraphs, there is no one right way to block your script. A good indicator of whether or not a block is a meaningful block is whether or not you can write a simple comment for that block.

Block comments should describe what you *intend* the code to do, not what it does in detail. The former can (and should!) be done even before you write the code (like unit tests discussed in defensive programming). These comments will not only be concise, they can actually help you write better code. Comments that describe how the code works in detail are simply repeating the actual code in words, and probably more clumsily. This is redundant; don't do it.

In-line comments

Complex lines of code should include in-line comments to help the reader understand what the line is doing.

```
z <- (x + y) / 2 # Average x and y
# Average of three averages
weird <- (((k + j) / 2) + ((x + y) / 2) + ((b + c) / 2))/3
```

Notice there is a limit to how much these in-line comments can help. They successfully translate the code into English (as opposed to simply repeating it), but absent any context, we can't understand what is happening, or why. That is what the block comments should provide.

Writing readable code

In-line comments make complex code readable, but simple code that doesn't need comments is better by far. Whenever possible write Really Obvious Code (ROC). Tips for writing ROC:

- **Use good names** Use function and variable names that are self-explanatory. For example, `random_locations_of_N_spiders_in_a_box.R` is a much better function name than `eek.R`. Don't worry about forgetting such a long name, or even typing it in. In RStudio, you can simply type `random[TAB]` to see a list of all functions you've defined whose name starts with `random`, so the extra characters don't waste time.
- **Do one thing at a time** This:

  ```
  foo <- make_a_foo()
  king <- crown_a_foo(foo)
  dethrone_a_foo(king)
  ```

  is much more readable than `dethrone_a_foo(crown_a_foo(make_a_foo()))` or even `make_and_crown_and_dethrone_a_foo()`. Each line is a sentence, and when you do many things on the same line, you run the risk of run-on sentences.
- **Don't use magic numbers** When programmers talk about magic numbers, they don't mean 7 (necessarily). They mean any number that's just sitting in your code, unnamed. This is similar to "use self-explanatory variable names." `area <- 5 * 3` is not as clear as `area <- width * height`.

Data Challenge

For this challenge, we will be using the paper Jager, Leah R., and Jeffrey T. Leek. "An estimate of the science-wise false discovery rate and application to the top medical literature." Biostatistics 15.1 (2013): 1-12. The paper can be found in the `readings` folder.

This tutorial has been focused on making your code reproducible and sharable with the wider world. Many journal now *require* that you submit your code and data when you publish your paper. Jaeger and Leek (2014) is one such article. While an earlier article claimed that over 80% of medical results were probably false discoveries, this work uses a slightly more sophisticated analysis of the p-values reported in the medical literature, to determine whether the observed distribution of p-values is significantly different from those drawn from a uniform distribution. Reassuringly, the authors find that the false discovery rate is lower (~20%), but still troubling, across many medical journals.

1) Use the published code in Jaeger and Leek (2014) to reproduce the reported mean and standard deviation of the FDR of *The Lancet* to 4 significant digits. Note that you should *not* need to scrape PubMed to do this.
2) Use the scraped abstract data provided in `data` and use the authors' code to compute the FDR for the journal PNAS. Find the mean and s.d. of the FDR for this journal.
3) Bonus question: use the authors' code to first scrape PubMed for abstracts, and next compute the FDR for the Journal of Psychiatric Neuroscience (PubMed abbreviation: J Psych Neuro). You will find a bug in the code that you will need to hunt down and fix.
4) Bonus question: Find the part of the code where the authors' initialize their random number generator.

Go to this link to submit your team's answers.