Reproducibility of data analysis

Stephanie Palmer & Stefano Allesina & Graham Smith

September 12-14, 2017

Contents

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.

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:

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

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.

What is reproducibility?

Here we'll consider three levels of reproducibility:

0. Accessibility
1. Replicability
2. Reusability
3. Extensibility

These are hierarchically ordered in necessity, difficulty, and goodness.

Reproducible code must be accessible to other researchers. However, accessible code is not necessarily reproducible in any sense. If the code is incomprehensible OR doesn't run, then you may as well not have published anything at all.

The first real level of reproducibility is replicability: Does your code run and produce the result you said it would produce? If yes, then at least your result is replicable. If no, then you haven't done science, you've done magic. In an ideal world, your reviewers would catch this.

Most scientists would probably aspire to reusability, the second level of reproducibility: can another scientist take your code and apply it to their own data? In order to apply your code correctly to their own data, another scientist will need to understand your code quite well. Therefore reusability requires more understandable code.

Finally, extensibility requires that your code be reconfigurable to address either different questions or different data. This is fundamentally different from the preceding, and is often beyond the scope of scientific inquiry. If you've written extensible code, you've made a tool, so this is more akin to methods development.

Accessibility

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.

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

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.

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
- `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`: *v.* switch branches.
- `pull`: *v.* download the latest version of the project from the remote repository.
- `push`: *v.* upload your latest local commit to the remote repository.
- `init`: *v.* Create an empty repository.
- `clone`: *v.* Create a new local copy of a remote repository.

Exercise 1: Your First Repo

1) 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 `Documents` or your equivalent. Note that you cannot put a `git` repository inside another `git` repository.

2) 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..." and save it.

3) 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.

4) Stage the change.

5) Type a commit message (e.g. "Began Essay on the Origin of Species") and commit the staged change.

If you use `git` in the wild, you'll find that staging and committing is 90% of what you do.

Remote repositories

For the remainder of the tutorial, we'll be using a more complicated example based in code hosted on GitHub. So far, we have been working with a local repository, meaning the repository is hosted only on your computer. Usually, you will also want to keep a copy of the repository online, called a "remote repository." With a remote repository, you can collaborate with others, sync your code across multiple machines, and back up your code.

The most popular option for hosting remote repositories is GitHub. As a 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, 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 tracking a remote repository, you `pull` the most recent version from the server to sync your local copy of the project to the most recent version. When you are done working, you `push` your commits to the server so that other users can see them.

Daily Workflow

1) Pull any new changes from your collaborators
2) Work on your project
3) When you've done something meaningful, stage your changes
4) Commit your changes, writing a meaningful commit message
5) Repeat steps 2-4
6) When you're done for the day, or when you've finished code that you want your collaborators to be able to access, push your changes to the remote repository (e.g. on GitHub).

Clone a Repo

[TODO: common use case: use someone else's code]

[TODO: clone LotsOBugs repo]

Replicability

So far, we've covered making your code accessible, whether to yourself in the future (version control) or to other researchers (*remote* version control). But accessibility is only the first step. To go further, we need to make sure other researchers can understand our code.

In order learn how to write understandable code, we're going to try reproducing some figures ourselves. After cloning the LotsOBugs repo, you have a copy of all the code used to make some figures you'd like to replicate.

Understanding the Code

Code should be understood from the top down.

Read the README

At the top level, every project should have a file called simply "README." This file should explain the purpose of the project and direct the reader to important files (e.g. the script that runs the analysis).

In this project, the README very thoughtfully explains where you can find the code that made each figure.

Read the block comments

At the top, every file should have a block comment describing the purpose of that file. Similarly, every block of code should have a comment describing its purpose. You might ask what we mean by "block." Blocks are like paragraphs. There isn't any rule as to what constitutes a block of code. Loosely, if you can write a concise comment describing the action of some lines of code, that would make a good block. So, rather circularly, a block is a set of code that has a block comment. Separate blocks by empty lines.

In this project, the block comments provide a nice way to quickly find the code responsible for the figures in question.

Read the inline comments

Inline comments are the most specific of comments. As a beginning programmer, you should probably use them liberally, to make explicit to yourself what each line does. However, in general inline comments should be used sparingly, if at all. Generally, if the code in a single line needs a comment to be understandable, then it's too complicated.

Read the code

While comments are useful and necessary, you should always strive to write code that is understandable by itself. Here are a few rules of thumb to help you write understandable code:

- **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`.

Exercise 2: Rename variables

TODO: Maybe move the below to the README?

A statistical interlude: 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.

Data Challenge

TODO

Make a pull request!

It should do X, Y, Z (tasks for 8 people)

The requirements are:

- It should include a commit from every group member.
- There should be at least one merge (either due to merging branches, or because of a merge conflict)