Analytical Workflows

All course notes strung together

November 26, 2020

Structuring your project

The general recommendations of this section in regards to establishing a consistent structure for your project should apply whether or not you plan to use version control software to manage your project or not. For example, the recommendations apply equally if you plan to use `Dropbox` or the equivalent (which you should *most definitely* be using if you're not going to use version control software).

Developing a project mindset

Like many grad students, I finished my thesis with

- one master folder called `Data` containing a bunch of sub-folders containing the various data sets (mostly Excel and CSV files) and some relational databases (Microsoft Access) that I'd collected or collated over the years;
- one master folder called `Rcodes` containing a bunch of sub-folders within it (each with a different project (chapter) and/or set of analyses of some set of my data);
- one master folder called `Mathematica` that similarly contained a bunch of sub-folders for various projects;
- one master folder called `Manuscripts` that contained all the papers and chapters I'd attempted or completed;

and a bunch more similar folders all variously named and "type-specific" within my overall `Research` folder. You might currently have something similar for your thesis work (Fig. 1).

Turns out that's a poor way to organize your work for a variety of reasons, including the ease of backing-up new data and code; the ease and efficiency with which you might expand, modify or branch off of your prior work; and even the simple reproducability of past work (by yourself down the road, or by someone else for whom you'll have to pull together all the necessary parts).

I now organize my work using a *project mindset*. I don't do this for each and every project idea or analysis I try out, but I do use it for "definitely doing this" (i.e. planned-out papers (thesis chapters)) and collaborative projects. By project mindset, I mean that (almost) everything associated with a given project is contained in one folder. I do still use a combination of `Git` and `Dropbox` to organize my projects based on my plans for projects[1] and collaboraton needs, but within each of `Dropbox` and `Git` I have my project folders organized within a master folder.[2]

All that said, defining "a project" can get difficult (esp. within your thesis work), so a fair bit of forethought is often needed. We'll definitely talk about this as a group in class. It's not trivial, but becomes quicker with time and seeing what others do.

> *Ben adds:* I also use project mindset to organize my folders. Something I like about project mindset is that it encourages what you might call *deliverables-based* thinking. By identifying and naming the "definitely doing this" projects, I am encouraged to consider my priorities, both within and among projects.
>
> For each project I am forced to think clearly about what the project is fundamentally about by having to name the folder. Asking "what should this project folder (or Git repo) be called?" (and insisting on an *informative name*)[3] is pretty close to asking "what is this project about?" So a project mindset supports clear thinking.
>
> I also find that a project mindset promotes better time management. For instance, all my project folders are contained within three superfolders: Active, Complete and Archive. The

[1] For example, whether I plan to make the entire project publicly available.

[2] Note that you're asking for trouble if you put a `Git` folder within your `Dropbox` or `GoogleDrive` folder, or vice versa.

[3] Naming conventions are something we'll come back to when we talk about good coding practices.

folders within Complete are named with dates and brief titles of publication. The Active folder contains stuff I am working on right now, that has not "shipped" yet. Archive is for stuff that is on the back-burner.

In this setup, the project folders within the Active folder - each with a name that reminds me of the objective for that project - becomes a kind of high-level to-do list. The goal is to be able to one day drag those folders from Active to Complete. Crucially, if the Active folder gets too full, I know I will not be able to do succeed in moving project folders because my attention has become too divided. So then I ask myself which are the most important few projects to me, and drag the rest to the Archive folder. It's not that I can't do them later. It is just recognizing that (a) they are not done yet, and (b) they are not the first, second or even third priority. If both (a) and (b) are true, into the Archive folder they go! OK, back to Mark, and the structure of an individual project folder.

Structuring your Project folder / Repository

For most projects, within each project folder, I usually have the following sub-folders:

`data`	the original (and cleaned) data required for the project
`code`	all the scripts needed to perform the analyses
`results`	all the output of the analyses
`biblio`	bibliographic files
`figs`	final figures (and tables) that go into the manuscript
`manuscript`	manuscript(s) derived from the project
`pdfs`	collection of relevant papers, manuals, etc.

The first three (`data`, `code` and `output`) are definites for any project (repository) that I plan to make public (e.g., on GitHub). For such public repositories, I do not include the other four folders. I use `figs`, `manuscript`, `biblio` and `pdfs` for manuscript-writing "projects" (which I place in an `InPrep` manuscripts-only sub-folder within my master `Git` folder). The contents of `figs` differs from the rough-and-dirty figures I save into the `output` folder. Sometimes `tables` get their own folder. Within `code` I might have an `R` and a `Mathematica` folder, as relevant. Note that `data` contains both the un-

touched original data[4] and the clearly-identified cleaned or otherwise derived data (for ease of subsequent access).

> *Ben says*: My sub-folder structure is similar, with variations depending on the project and preferences. For example, my reference manager of choice keeps all my pdfs in one place, so instead of having a pdfs folder in each project folder, I have "folders" for each project within my reference manager. Either way, the same goal is achieved: a logical hierarchical structure that makes it easy to find and keep the various pieces of a project. Back to you Mark.

Most of the time when using `Git` you'll have one *repository* associated with each one of your *projects*. A *repository* is thus synonymous with a *project folder*. When using `Git` you'll also have a few other files within the repository: a `README.md` file and a `.gitignore` file. If you're using `R-Studio` in combination with `Git` (as we will in this course), then you may also have an `.Rproj` file in the repository. It's good practice to have a `README.md` file in each of the "important" folders, describing what in them.

[4] Leaving your original data as untouched as possible is something we'll come back to during Coding Best Practices

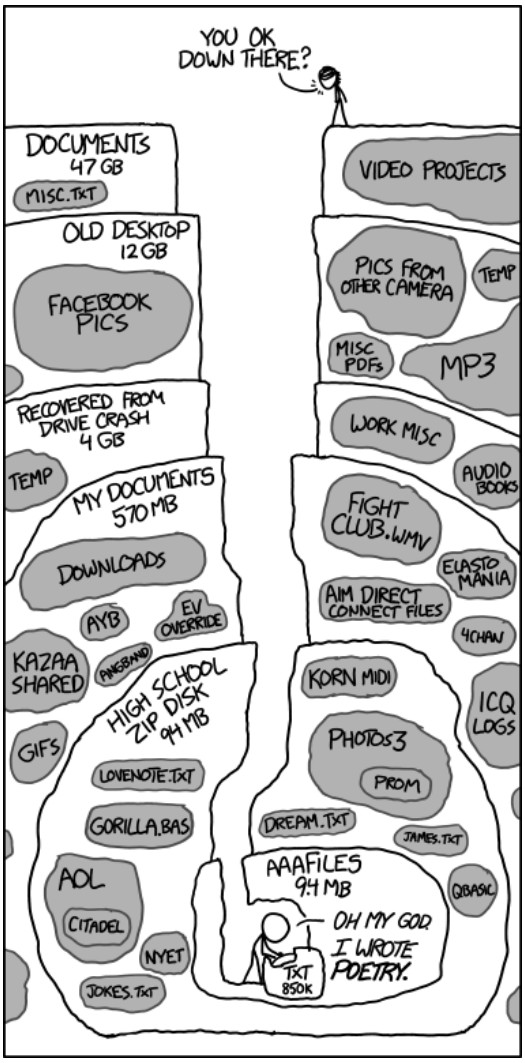


Figure 1: Old files (source: http://xkcd.com/1360/)

Getting started with `Git` & `GitHub`

Contents

What is version control? **2**

Git **2**

Github **2**

Installing and configuring Git **3**

Repository setup **3**

R-Studio and Git GUIs **5**

Git workflow **5**

What is version control?

Version control software allows you to save and provide meta-information on any and all changes you make to a set of files and directories. It allows you to more easily document changes and the evolution of your files, find and correct bugs that have crept into your code, and revert back to old versions when needed. By using version control you'll know what, when and (hopefully) why changes were made. If you're using the software collaboratively, you'll also know by whom changes were made. Backing-up is virtually implicit when you also use a networked (off-site) server as a host for your repository.

We'll be using `Git` as our version control software. There are others out there (e.g., `Subversion`). We'll be using `Github` as our host. There are others out there (e.g., `Bitbucket`).

Git

`Git` was developed by Linus Torvalds (the "Linu" in Linux). Most of its features are accessed by command-line and are intended for large-scale collaborative programming and software development purposes. Of course, we'll be using `Git` for much smaller, specific projects, thus we won't bother with many of these feature. We'll also interact with `Git` using GUIs (graphical user interfaces, e.g., `R-Studio`, `Sourcetree`) rather than command-line.

Github

`Git` stores a complete copy of the project on your local machine, including all its history and versions; no centralized server is necessary. However, if you want to collaborate with others, have multiple computers with which you'd like to work, or want to create true back-ups of your project, then you'll also want to use a server on which to host your repository. Fortunately, there are a number of free providers.

For `Git`, the primary options are `Github` and `Bitbucket`.[1] The former is more developed (more bells and whistles), is currently more widely used, and is perhaps a little easier to work with. The two don't differ all that much

[1] I suggest creating accounts with both `Github` and `Bitbucket`. I use them for different types of projects, e.g., public verus private, as needed.

except in one regard: the number of free versus public repositories. While `Github` has a limit on the number of private repositories, `Bitbucket` has a limit on the number of collaborative projects (having more than 5 collaborators). (There are perks regarding the number of repositories you can have if you sign up using an academic email address. See https://help.github.com/en/articles/applying-for-an-educator-or-researcher-discount)

Installing and configuring Git

See the `README.md` of our very first class for installing `Git`.

After installation, there's a little (minimum of) command-line configuration to perform. On a Mac, open a `Terminal` window and type in the following:

```
$ git config --global user.name "Mark Novak"
$ git config --global user.email "Mark.Novak@oregonstate.edu"
```

Some of you might want to set up an editor to be used in git, for example:

```
$ git config --global core.editor atom
```

You can check to ensure that these commands went through and see what other things you might want to configure using

```
$ git config --list
```

For more, or if you're using Windows, see

https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

Repository setup

There are command-line methods for doing everything we're going to do below. Indeed, command-line is the default way to interact with `Git`.[2] Instead, we're mostly going to make use of the tools made available through `Github`, starting with the very first step of initiating a repository and getting some minimal information associated with it. That is, rather than creating and setting up a repository on our computer by command-line and then connecting it to `Github`, we'll create and set up the repository on `Github` and then clone it to our local master folder of projects.

[2] See last page for a cheat sheet.

Simply login to your `Github` account, click on "New Repository", and follow the instructions.[3] These should include options for private vs. public (the latter is preferred for this class[4]), initializing with a `README.md` file (which you *should* do), and adding a `.gitignore` file (which you *should* also do).

The `README.md` file in the main repo folder is the first file that anyone will see when they inspect your repository (assuming it's public). At minimum, it should give an overview of what the project is about and what the various parts of the project structure are. We will learn to use Markdown to write and edit `README.md` files later in the course, so for now just leave it as is.

The `.gitignore` file contains a list of all the files that you want `Git` to ignore (i.e. not monitor for changes). Selecting `R` from the dropdown list will auto-populate a bunch of it for you. Later in the course, we'll also add the extensions for all the temporary files that LaTeX produces when compiling.

You should now see a new webpage – your Repo page – that shows you what's in your repository. For now it contains only the `.gitignore` and `README.md` files, the latter of which has its contents displayed.[5] As I said earlier, there are a lot of bells-and-whistles at your fingertips here. We'll ignore them for now, but feel free to explore! You *could* start dragging-in directories and files into your browswer view to add them to your repository, but we're *not* going to do that. Instead, we're going to `clone` this repository to our local machine, then add our various project sub-folders to it (e.g., `code`, `data`, and `results`), and go from there.[6]

To clone the repository, click the green `Clone or download` button and copy the provided URL. There's a few ways to clone your repository to your local machine. Your preferred method depends on how you're likely to interface with `Git`. You could:

1. use command-line to clone. Open `Terminal`, `cd` into your `Projects` master folder, then type `git clone` followed by the URL you just copied;

2. use a visual `Git` GUI client to clone the repo;

[3] When doing so, be sure you're in your own user environment and not inside our Analytical Workflows organization.

[4] If at all possible, please pick public (for this class) and switch to private afterwards. Otherwise, please add me as a collaborator so I can see your repo.

[5] There are actually other files in your folder as well, but their hidden by default.

[6] Note that empty folders will not be monitored by `Git`; they need to contain something.

or, if you're primarily going to be using this repository to keep track of an R-based project using R-Studio:

3. set up a "project" within R-Studio first and provide it with the URL during setup. It'll then clone the repo for you.

R-Studio and Git GUIs

I use Git for both R and non-R (e.g., Mathematica)-based projects. Only R-Studio has integrated Git functionality, so I use a visual Git GUI client (e.g., Sourcetree) for some projects because I haven't yet bothered to memorize the Git command-line commands. Since most of you are probably using R, it's probably worthwhile to use R-Studio's Git integration feature.

You'll first need to tell R-Studio that you have Git installed, so go to its Preferences, select Git/SVN and fill in the details: either click on the Help link or go to http://r-pkgs.had.co.nz/git.html to see what to do.

Now create your "project" within R-Studio by selecting "New Project". Select Version Control: Checkout a project from...repository, select Git, and fill in the details including the URL you got from Github. The directory in which you place your repository should be your master folder. R-Studio will "restart" and then you'll be in your project (as evidenced by its name appearing in the top-right of the interface). Clicking on the Files tab will show you what's in the repository (which should, at present, be: README.md, .gitignore and the newly created .Rproj file).

You may now create (or move in) all your project sub-folders.

Git workflow

Before proceeding, jump over and do the **required reading:** https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F. Then come back here.

Basically, files (or directories) exist in one of four states of a life-cycle: *untracked*, *staged*, *unmodified*, or *modified* (see Fig. 1). The standard workflow is thus:

1. Add or modify some files;
2. Stage the new or modified files;

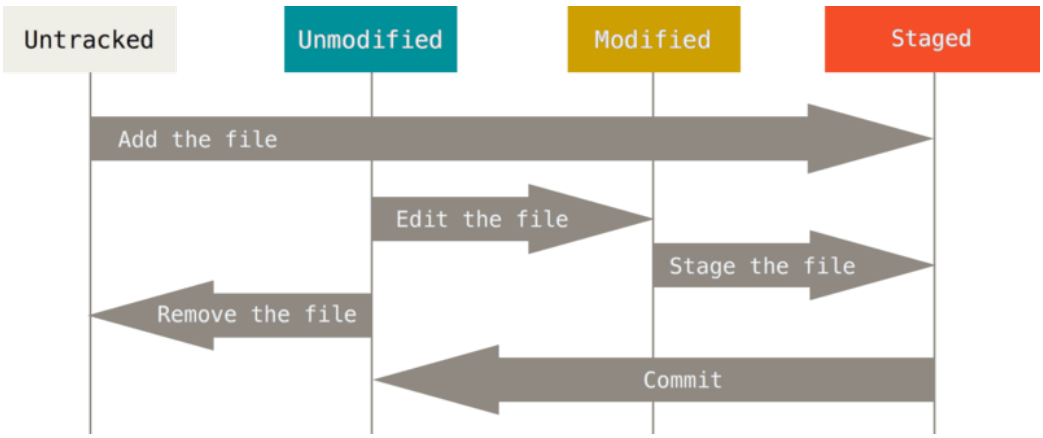


Figure 1: The `Git` life-cycle.

3. Commit the changes (moving them from the Staging Area to the "memory" of the repository);

4. Repeat.

Your motto for using `Git` should be "*commit early, commit often*". Every time you add or remove something from your project (i.e. a file or a chunk of code), you want to commit those changes. Ideally, each commit should correspond to a "logical unit", one that you are able to describe in a few words (e.g., "Add function to perform resampling"). Choosing when to commit is quite important, especially when you're debugging code. For example, if you've discovered your code has two bugs then you should commit each one of the fixes seperately, not together. That way you can undo either fix independently if, for example, you messed up in one of your fixes or your fix created a different bug somewhere else in your code.[7]

`Git`-GUIs provide visual interfaces for viewing your files, staging area, and commits. Within `R-Studio` (assuming you have your `R-Studio` project opened), looking at the `Git` tab will show you a list of all the files (and directories) that have been changed, removed or added to the project since the last time you committed. Clicking on the check-boxes associated with each file will add them to the staging area. Clicking on `Diff` or `Commit` will open up a new interface (the staging area). In the top-left corner you'll see a

[7] We'll talk about using "branches" to reduce the incidence of problems down the road.

list of the staged files. Selecting one of the files will bring its contents up in the window below which highlights the text that has been added (in green) and removed (in red). Select all the files you want to commit, stage them, add a commit message to the top-right window, and commit. You've now updated your local respository.

How to write good commit messages is a topic unto itself! For now, the only thing we'll say is that a properly-formed `Git` commit subject line should always be able to complete the following sentence: If applied, this commit will *your subject line here*.[8]

Clicking on `History` (top-left) will show you all your past commits.

Remember, "*commit early, commit often*" and provided concise and informative commit messages (Fig. 2).

The final thing to do (not necessarily following each commit) is to `Push` your commit to `Github`. `Pull` does the opposite: bringing commits that have been saved to `Github` (by others, or by you on a different machine) to your local machine. To reduce the likelihood of creating conflicts, *always* pull before you start making edits, commit and push. There are of course ways of dealing with conflicts and merging files (that will arise if collaborator(s) are working on the same file, for example), but pulling first will do a lot to avoid unneccessary hassle.[9]

So again, our basic recipe is: *pull, create/edit, stage,commit, push, repeat.*

[8]Every commit message must at minimum have a "subject line". In fact, the subject line could be the only thing in your message. However, you can also write a whole lot more if you'd like, a paragraph even, by adding a blank second line between the subject line and the rest of your message. For a great post on writing commit messages, see https://chris.beams.io/posts/git-commit/.

[9]We'll learn about merging and conflict resolution later in the course.

git cheat sheet

learn more about git the simple way at *rogerdudler.github.com/git-guide/*
cheat sheet created by Nina Jaeschke of *ninagrafik.com*

create & clone

create new repository	*git init*
clone local repository	*git clone /path/to/repository*
clone remote repository	*git clone username@host:/path/to/repository*

add & remove

add changes to INDEX	*git add <filename>*
add all changes to INDEX	*git add **
remove/delete	*git rm <filename>*

commit & synchronize

commit changes	*git commit -m "Commit message"*
push changes to remote repository	*git push origin master*
connect local repository to remote repository	*git remote add origin <server>*
update local repository with remote changes	*git pull*

branches

create new branch	*git checkout -b <branch>* *e.g. git checkout -b feature_x*
switch to master branch	*git checkout master*
delete branch	*git branch -d <branch>*
push branch to remote repository	*git push origin <branch>*

merge

merge changes from another branch	*git merge <branch>*
view changes between two branches	*git diff <source_branch> <target_branch>* *e.g. git diff feature_x feature_y*

tagging

create tag	*git tag <tag> <commit ID>* *e.g. git tag 1.0.0 1b2e1d63ff*
get commit IDs	*git log*

restore

replace working copy with latest from HEAD	*git checkout -- <filename>*

Figure 2: Don't let this happen! (source: http://xkcd.com/1296/)

Best practices

October 13, 2020

Contents

Motivation **2**

Principles **3**

Writing modular code **3**

Project modularity 3

Absolute versus relative paths 5

Starting fresh each time 6

Within-script structure 7

Using functions 9

Unit testing 10

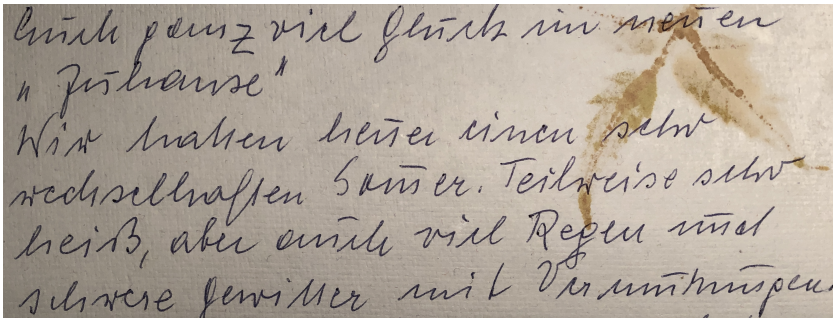
Writing readable code **12**

When writing code, though shalt also not... **13**

Motivation

My grandmother wrote in cursive with near perfect internal consistency. Like others in her generation, she was good at it, having been required to do so since early in her education. Her language was German. German has a rather different sentence structure than does English. It's easy to write an entire paragraph with a single sentence that uses nothing but commas for punctuation yet remains perfectly clear and understandable. German also has easy-to-create compound words, unlike English. My grandmother spoke and wrote with one of the many dialects that exist in Austria. Not a strong dialect by any means – easy enough for any German speaker to understand. It has a few different words, but otherwise adheres entirely to the standard German alphabet, which is the English alphabet with only one additional character (ß), and three umlauted vowel forms (ä, ö, ü).

I myself have spoken German all my life (since before I learned English) and have no problem understanding most dialects. Nonetheless, reading my grandmother's writing takes me *forever*. Some words are simply unintelligible for me. Can you read – let alone understand – any of it? Virtually incomprehensible, isn't it? Consider what it would be like to try read a whole letter, or even a whole book written in this manner (in English)?



Even if you're relatively new to coding, every one of us has learned or developed habits and styles that make our code more difficult to understand. Moreover, research shows that those habits reduce our potential productivity, introduce errors, and even influence our ways of solving problems.

Today's class topic is about exposing you to a variety of recommendations for improving your code. Admittedly, I too am still working on better-incorporating these recommendations into my own habits. That said, there are a lot of recommendations for "best" and "better" practices out there, not

all of which will work for you, your type of research, or your specific project. But be careful not to dismiss them too quickly. The reality is that for any non-trivial task, most of your time will be spent de-bugging your code, not writing it. Putting in the time and effort up front to learn or improve upon your best practices will pay off many-fold in the future.

Principles

Writing code is in some ways simple. Learning to write in a computer languange (or flavour) is nothing more than learning to put together the words of a human language (or dialect). It takes only practice and repetition to learn the vocabulary and how words fit together to make sentences. However, to become fluent (and hence communicate effectively) requires learning to use grammar and style.

Like any written language, code has hiearchical units that are integral to grammar and style. A useful way to think about code is therefore as follows:

Programming	Language
Scripts	Essays
Sections	Paragraphs
Lines breaks	Sentences
Parentheses	Punctuation
Functions	Verbs
Variables	Nouns

Thinking about these parallels should allow you to recognize two over-arching principles that will, when practiced, greatly improve your code and code-writing efficiency: writing hiearchically-*modular* code and writing *readable* code.

Writing modular code

Project modularity

The highest level of modularity relates to the *project mindset* with which we started this course. If your experience learning to code has been anything like mine then, either when you first started coding or when you wrote homework assignments for Stats class, the code you created for a given project

probably consisted of one long file. That'll work fine for small (tiny) projects or homework reports, but probably not for a project that will result in a paper or thesis chapter that contains some combination or subset of data, analyses (whether statistical or non-statistical modeling), and data- or results visualization. So, to refresh our memories, you should give your (`Git`) `project` folder a useful structure.

Your `data` folder should contain all the data you need for the project.[1] Most of the time, this will include both the *original* unedited raw data files (regardless of their format) as well as *derived* data files (that you have used code to produce). Raw and derived data must be clearly distinguished or placed in seperate folders (e.g., `DataRaw` and `DataDerived`). No files should be duplicates or derived copies (versions) of each other (see Fig. 2); remember that versions will be tracked by `Git`.

Your `code` folder contains your various script files. For anything but the smallest projects there should be multiple files that contain scripts for performing different tasks. For example, for a simple project you might have:

`data_prep.R`	Script that pulls in the original data, pre-processes them (rearranges, correct errors, standardizes names, splits up or merges different data sets), and spits them back out in a "clean" format needed by your subsequent analyses;
`my_functions.R`	Script containing the functions you have self-defined to perform your analysis;
`analysis.R`	Script that sucks in your "clean" data, performs your analysis (using self-defined and package functions), possibly makes some quick-and-dirty figures along the way, and exports the results to your `output` folder;
`final_figs.R`	Script that sucks in the results of the analyses to produce the final figures (or tables) for your manuscript.

In general, you don't want any script to become unwieldy. Thus, for larger projects, you will likely have multiple scripts within each of the above categories. For example, if you have several different dataset types, you might need to have several `data_prep` scripts (e.g., one for each type). Similarly, if you have a large number of self-defined functions, you'll probably want a function `library` folder with each function in its own script file. If you end up with a a sizable number of scripts to perform sequentially from start

[1] Along with their accompanying meta-data!

to finish, then you will probably also want one additional master `RunMe.R` script, placed within the main `code` folder[2] that sources each of the other scripts in the appropriate order to recreate everything from start to finish.[3]

The key utilities of separating things into such modular units at this top-level are (1) *readability* and, for functions in particular, (2) *unit-testing*. Readability means that it's easy for anyone (including you in just 6 months time) to figure out where things are being pulled from, what's being done, and where the output is going. Moreover, no individual script is overwhelming to look through and figure out. The idea behind unit-testing is to write independent tests for the smallest units (chunks) of your code. For example, whenever you write a function, you also write an associated small piece of code that tests it (before the function is used in your primary analysis). That way, whenever you modify your code or function(s), it's easy to ensure that everything is still returning the correct output. We'll come back to unit-testing below.

Absolute versus relative paths

In order to employ the principle of modularity you need to know how to navigate between your various folders and call upon your various datasets, scripts and results. Within an `R`-project, for example, you have a choice between using the repository folder as your working directory or setting your working directory to be the directory in which you have saved your scripts (e.g., your RunMe script). Your scripts will therefore need to access data and source scripts that are located in other folders. You want to set these locations as generically as possible so that you (or someone else) can move the entire project folder to a different location without destroying the workflow, for example. You never want to require setting the working directory more than once in a work session. Nor do you ever want to "hard-write" the locations of data and scripts.

The way to do this is by specifying *relative* rather than *absolute* (full) directory paths. For example, assuming your working directory (i.e. the directory you're curently in) is the `code` directory, rather than specifying the location of your data using its absolute location:

[2] alongside its explanatory `README.md`

[3] If your project gets really hairy, you might even consider using a MakeFile (https://cran.r-project.org/package=MakefileR) or using Drake (https://github.com/Factual/drake) or the like.

`read.csv(file='/Users/marknovak/Git/MyProject/data/data.csv')`[4]
you should instead use the relative path:

`read.csv(file='../data/data.csv')`

The latter takes you up one level (out of `code` into your main project folder) then into the `data` folder and to your data file. Each repetition of '`/../`' will move you up one level in the folder structure. To pull in (source) your self-defined functions, use

`source(file='my_functions.R')`

since you're already in the `code` folder. **Note:** Mac (Unix) and Windows (DOS) use forwardslashes and backslashes differently! Mac uses '`/../../file.R`' (forwardslashes) while Windows uses '`\..\..\file.R`' (backslashes).

Starting fresh each time

You should write your code so that you can start each work session fresh. That is, you don't want to (nor require needing to) leave anything (e.g., variables, functions, packages) hanging over from a different project or previous work session. You may even want to avoid having anything hanging over from a previously-run script. It may seem convenient and faster to save your work session, but it will only cause problems, bugs, or worse (faulty results).

There are two basic approaches to starting fresh: The first is to clear your workspace with `rm(list=ls())` wherever appropriate; at the top of your `RunMe.R` script, for example. Similarly, you might place `rm(list=ls())` at the top of your `analysis.R` script since your "cleaned" data was saved and can (will) be reloaded, leaving all the temporary variables created in the cleaning of the data unnecessary. In contrast, you wouldn't put it at the head of your `my_functions.R` script. The second approach considers the use `rm(list=ls())` insufficient because all it does it clear out your workspace; it doesn't clear out loaded packages, it doesn't clear out any altered global options (e.g., `options(stringsAsFactors = FALSE)`), and it doesn't reset the working directory (potentially making the project less reproducible by others).

The over-arching rule is to *never* save your workspace for reuse. (Turn off the end-of-session prompt in your `R` or `RStudio` preferences.) If something takes a long time to create, use `save()` to place it into a `.Rdata` file which

[4] In Windows it might look more like `file='C:\\MyDocuments\Git\MyProject\data\data.csv'`

will be quick to `load()`. You could even place this file into a `tmp` output folder that gets deleted at the very end of your `RunMe.R` script.

Within-script structure

Similar to a manuscript, a typical script should consist of the following sections, each visually separated from the others:[5]

1. Start each script file with a preface that describes what it contains and how it fits into the project;
2. Load all required packages;
3. Source required scripts (`my_functions.R`);
4. Load the required data (or source the `data_prep.R` script(s)); .
5. Sections for the major parts of your analysis;
6. Export results section.

The last two may consist of just two sections or you may have export sections immediately following each major part of the analysis.

A simple script might look as follows:

```
1  ############################################################
2  # simulateLV.R
3  # Simulates the dynamics of a predator and a prey
       population according to the Lotka-Volterra model.
4  # The data produced will subsequently be used to test
       the performance of several population dynamic model-
       fitting routines.
5  ############################################################
6  rm(list=ls()) # clear workspace
7
8  ##############################
9  # Load libraries
10 ##############################
```

[5] In RStudio you can further take advantage of folding: https://support.rstudio.com/hc/en-us/articles/200484568-Code-Folding-and-Sections

```
library(deSolve)

##############################
# Source files
##############################
# None needed

##############################
# Define model
##############################
LVmod <- function(Time, State, Pars) {
   with(as.list(c(State, Pars)), {
      Ingestion    <- rIng  * Prey * Predator
      GrowthPrey   <- rGrow * Prey * (1 - Prey/K)
      MortPredator <- rMort * Predator

      dPrey        <- GrowthPrey - Ingestion
      dPredator    <- Ingestion * assEff - MortPredator

      return(list(c(dPrey, dPredator)))
      })
   }

##############################
# Define parameters
##############################
pars <- c(rIng   = 0.2,    # /day, rate of ingestion
           rGrow  = 1.0,    # /day, prey growth rate
           rMort  = 0.2 ,   # /day, predator mortality
           assEff = 0.5,    # assimilation efficiency
           K      = 10)     # mmol/m3, carrying capacity

##############################
# Simulate model
##############################
yini  <- c(Prey = 1, Predator = 2)
times <- seq(0, 200, by = 1)
out   <- ode(yini, times, LVmod, pars)
summary(out)

```

```
51 ##############################
52 # Plot and export data
53 ##############################
54 plot(out)
55 write.csv(out, file='../output/LV_out.csv')
56
57 ############################################################
```

Using functions

A key aspect of modularity is defining your own functions. Writing your own functions allows you to do many things, but the most important of these are readability (especially of the main code in which the functions are applied), simplifying de-bugging, and reducing the introduction of errors in the first place. Functions do that because they allow you to apply the D.R.Y. principle: "don't repeat yourself." More specifically, almost any set of commands that are repeated in two or more places should be converted to a function. Repeated use of copy-paste-tailor makes it extremely difficult (and error and inconsistency prone) to make changes or corrections.[6]

When defining a function, you should provide additional information that describes what the function does (in general terms), what inputs it takes, and what output it returns. If it takes a paragraph to explain what the function does, try choping the function up into smaller functions, or reorganize it so that several short explanations are possible.

A script for a function might look like the following:

```
1 ##########################################################
2 # Function to add stochastic noise to a time-series of
     population sizes.
3 ##########################################################
4 # Input:
5 #  x -- a time series of population sizes (vector)
6 #  noise_model -- gaussian (currently implemented model,
     default)
```

[6]The D.R.Y. principle applies to data and specified arguments and parameters as well. Every piece of data ought to have a single authoritative representation in your `data` folder, and any hard-coded parameter values should be defined exactly once to ensure that your entire project uses the same value, as appropriate.

```
7  #  sd -- the standard deviation of gaussian noise (
       default=0)
8  # Returns:
9  #  Vector of length equal to the input vector
10
11 add_noise <- function(x, noise_model='gaussian', sd=0){
12    noise_model <- match.arg(noise_model)
13    if(noise_model=='gaussian'){
14       out <- x + rnorm(length(x), sd)
15    }else{
16       out <- x
17       warning(''Original time series returned'')
18    }
19    return(out)
20 }
21
22 # Test:
23 # source('simulateLV.R')
24 # new <- add.noise(out, sd=1)
25 # par(mfrow=c(1,2))
26 #  plot(out)
27 #  plot(new)
```

Unit testing

After the function is defined, it is important to make sure it's working as expected! The easiest place to put those tests is immediately following the definiton of the function (just as done in the above example). Don't remove these tests after you've confirmed your function works as desired, just commented them out; you may want to re-test again if you make some changes to your function.

That said, notice that the unit test in the above example is a pretty poor one. All it does is help you (visually) ensure that the plot of the new data is different from the original data (and that the function returns meaningful-looking data to begin with). It does have some error-like catches built in, like returning the original data with a warning when `noise_model` is misspecified. But what if the standard deviation of the error was specified to be a large enough value that `out` contains negative values (i.e. non-sensical

population sizes)? What if there's a bug somewhere that causes the function to silently return `NA`'s? For such possibilities it's wise to spell out your expectations (both for the inputs and the output) and have your function issue a `warning()` (or even `stop()` excecution) when those expectations aren't met using internal `if-else` tests. Moreover, you should run your function on several test cases (especially extreme cases, in situations where they exist) where you know the correct answer. Getting into the habit of doing this for your self-defined functions and even at key steps of your analysis will save you time and headache in the long run (Fig. 1).

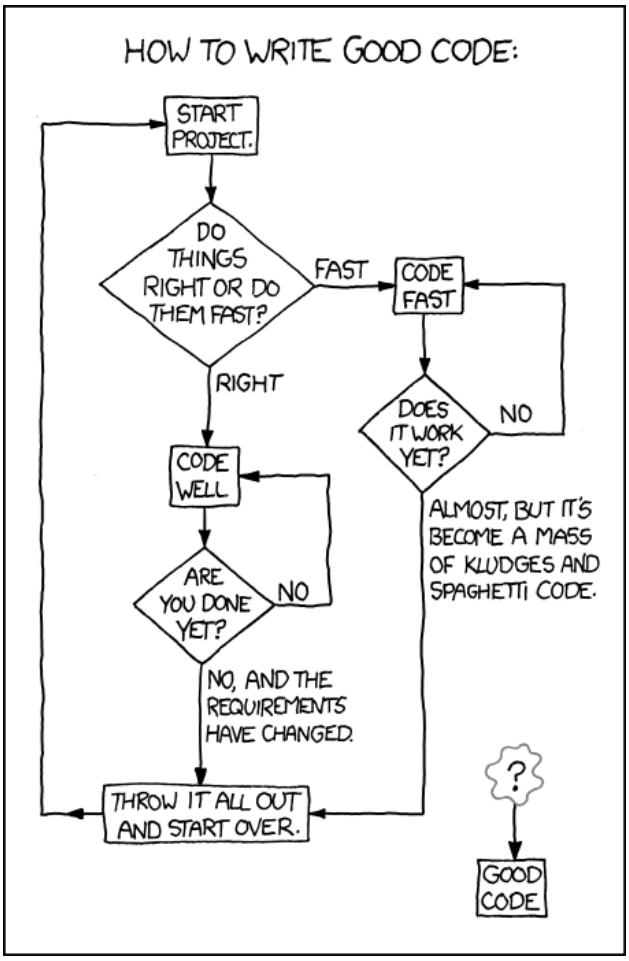


Figure 1: Good code? (source: https://xkcd.com/844/)

Writing readable code

There are a lot of summarized sets of recommended best-practices for writing readable code in general, and for `R` specifically. These include aspects of inline annotation (commenting), object naming conventions (for filenames, function names, and variable names), and syntax and grammar (spacing, indentation, etc.).

In regards to inline annotation, it's rare that code is commented enough. Believe me, you simply won't remember what certain key lines of code do after even just a few months of working on something else. Note, however, that comments which simply recapitulate code aren't useful. Instead of *how*, use annotation that explains *what and why*.

For code grammar and filenames (Figs. 2 & 3), read the Tidyverse Style Guide and Google's Style Guide amendments to it (**required reading**) before returning to continue reading here.[7] https://style.tidyverse.org https://google.github.io/styleguide/Rguide.xml

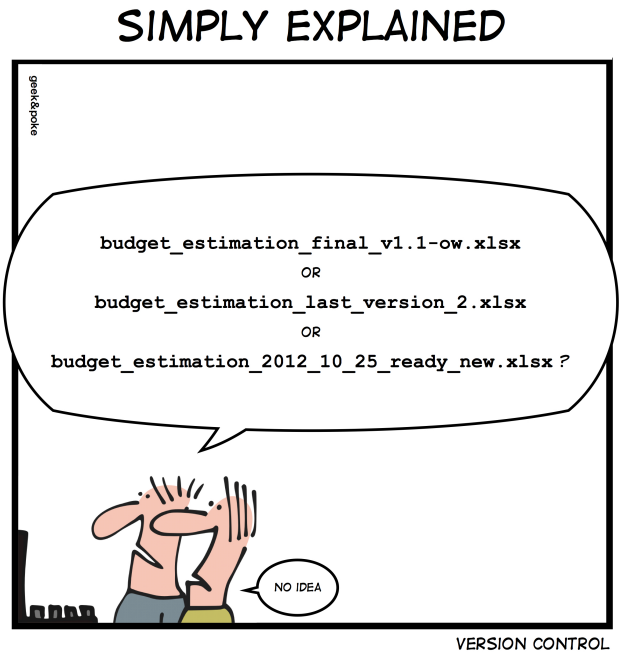


Figure 2: Your data files *will not* look like this; use `Git`!

[7] Curiously, I'm pretty sure that Google created their style guide first, but it then got superseded by the Tidyverse guide which Google now refers you to.

Figure 3: A comprehensive map of all countries in the world that use the MMDDYYYY date format; put thought into your filenames. (source: https://twitter.com/donohoe/status/597876118688026624)

Now, even before you become expertly practiced at writing beautifully-readable code (and assuming you're using `RStudio`), try the following.[8] Highlight some of your code and select `Code > Reformat Code` from the drop-down menu (or type `Shift + Cmd/Ctrl + A`)). `Reformat code` will try to alter your code to make it adhere to a style guide quite similar to (derived from) Google's style guide. You'll probably still need to do a little clean-up. You can also use `Code > Reindent Lines (Cmd/Ctrl + I)` to clean up nested loops, multi-line functions, and conditional (if-then) sections.[9]

When writing code, though shalt also not...

- copy-paste-edit (i.e. reuse and modify) the same code to repeatedly apply it to different subsets of data. Instead, you will convert your code to function(s) that can be applied to these data subsets.
- use `attach()` and `detach()` on your data. Instead, be explicit in naming and accessing data.frame columns.
- create large tables by hand. We'll learn LaTeX and export them instead.

[8] In principal you should really be doing this after initiating a new `branch` in your `Git` repository, but we haven't formally learned about branching...yet.

[9] For even more, try out the styler package (https://github.com/r-lib/styler).

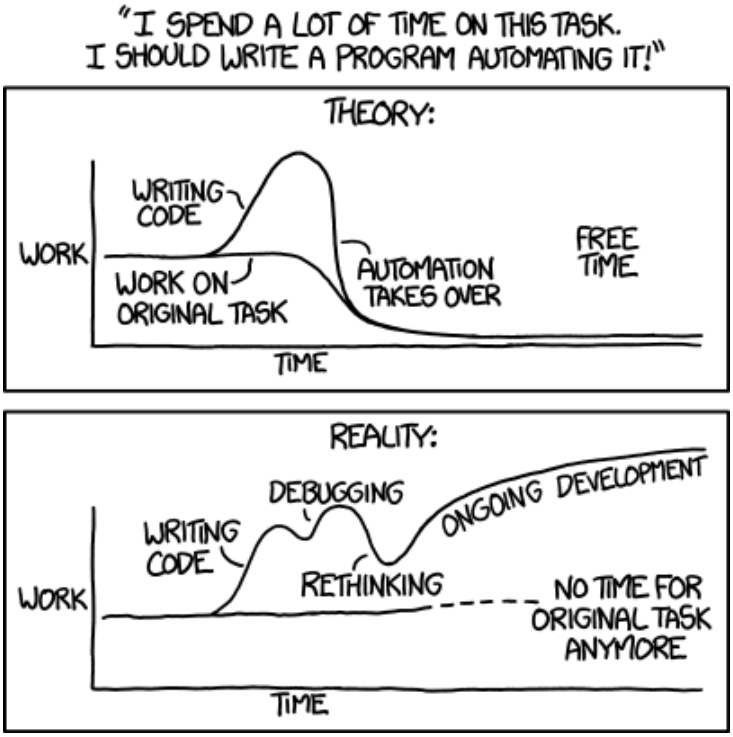


Figure 4: Admittedly, sometimes reality is indeed more like this. (source: https://xkcd.com/1319/)

More with `Git` and `GitHub`

October 20, 2020

Contents

What to (*not*) commit **3**
Untracking files 4

Branching **5**

Differencing & Merging **6**
`diff` for highlighting changes 6
`merge` for collapsing branches 8
Resolving merge conflicts 9
Pull Requests 9

Forks vs. clones & branches **10**

Going back in time **11**
Undoing commit(s) 11
Recapturing older commits 13

Project Management **13**
Issue tracking 13
Project boards 14

By now you've hopefully gotten into a new routine of working with `Git` (*pull, edit, commit, push*) and are doing well to remember the `Git` motto: *commit early, commit often*. Hopefully your repository is also looking nothing like Fig. 1. The goal for today is to extend our workflow by making it more flexible and adaptable to more diverse project structures and more general research contexts (such as collaborative projects). To do that, we'll learn the basics of a few more tools that `Git` and `GitHub` offer. Almost all of what we'll talk about below can be done using a `Git` GUI and on `GitHub`, but I might often describe actions using the command-line process.

Figure 1: The whole point of a version control system is to avoid your project folder devolving into this! (source: http://maktoons.blogspot.com/2009/06/if-dont-use-version-control-system.html)

What to (*not*) commit

Up to this point, under the guiding principle of reproducibility, we've implied (if not explicitly stated) that everything associated with your project should be kept in your version control respository.[1] That's not entirely incorrect, but it's also not entirely true. For example, `Git` will throw an overridable warning when you commit large files (currently $\geq$ 10mb), and `GitHub` will throw a fatal error when you try to push very large files (currently $\geq$ 100mb)[2]. The latter can happen with specialized data files (e.g., GIS shapefiles). `GitHub` also as a limit on the total size of any one repository (???mb). Sometimes it's therefore necessary to keep some files in a different directory outside of your repository (preferably tracked by `Dropbox` or similar cloud storage).[3] Sometimes it's possible to cut-up your data into smaller pieces. Other times, and more generally speaking, it's possible to reduce the size of a file by saving its contents in a different format, a plain-text format.

What I mean by plain-text format is most easily seen by comparing the contents of a data worksheet saved in a `.csv` file or text saved in a `.txt` file to the same data or text saved in, for example, a Microsoft Excel `.xls` or Word `.doc` file. The former are human-readable when opened and edited in even the most basic of text editors. In contrast, if you were to force open the `.xls` or `.doc` files in the same text editor you'd get a whole bunch of unintelligible and indecipherable computer symbols. That's because the `.xls` and `.doc` files are binary files and have a whole bunch of additonal information in them for interpreting, presenting and otherwise doing things (e.g., calculations) with your data and text that requires specialized software.[4] We don't want all that extra stuff for a few reasons:

1. your data should contain nothing but data;

2. we're going to do all calculations by script so as to make them reproducible and readable;

[1] It should go without saying that anything private (e.g., passwords) should never be put in a public repository, but be careful not to do that when we start submitting jobs to High Performance Computing clusters!

[2] If necessary, you could use the Large File Storage extension: https://git-lfs.github.com

[3] See back to our `Structured Projects` class.

[4] `.Rdata` files are also binary files that need R to open them.

3. your text documents should remain readable even in 10 years time and even after Microsoft Word has yet again updated to save things in a new way;

4. every time you open and save a `.xls` or `.doc` file, a whole bunch of that computer language information is changed, causing `Git` to unnecessarily track these changes;

5. we want to make it easy on us to visualize exactly what content has changed in a file since it was last committed (or between any two commits) using a `Diff` tool that we'll talk about below.

So challenge yourself to use plain-text file formats whenever possible. Plain-text files include `.csv` files, `.R` scripts, Markdown `.md` scripts, and LaTeX `.tex` files. Images (e.g., `.jpeg`) and movies (e.g., `.mp4`) are not.[5] Microsoft- and Apple suite files and the like are (mostly) not plain-text files.

Other types of files you should not commit are temporary files, such as `$.xls` files that are generated only when a program like Microsoft Excel is open, or anything but the primary LaTeX `.tex` (and perhaps the associated `.pdf`) file (because these all get generated each time you compile your document). You should also not commit your `.Rprofile` file (since it contains proprietary API keys). All these temporary file types can be added to your `.gitignore` file.

Untracking files

What to do when you accidentally commit a file you don't want `Git` to track (e.g., you forgot to add it to `.gitignore`), or if you change your mind about a particular file and no longer want to track it? If you're using a `Git GUI`, you can probably just right-click the file and select `Untrack` or `Stop tracking`. To remove the file from the staging area via command-line, use `$git rm --cached` *`filename`*. To remove the file entirely (i.e. from the repository), use `$git rm` *`filename`* (without the `--cached`).

[5] Their large file size also makes `Git` and `GitHub` a poor place to store large amounts of image and movie files.

Branching

So far, when we've committed to our `Git` repository, we've been doing so to its *master* branch. The *master* branch has been our only branch, and every change we've made to a file has been changed sequentially (i.e. has overwritten what was there before). In other words, our commit workflow has been pretty much linear.

But there are often times when you have an idea for doing something differently. You might have an idea for optimizing a section of code (e.g., by vectorizing the use of a function, rather than using a `for` loop[6]), or you might want to try writing a section or paragraph of your manuscript a different way[7]. In both cases you don't want to "break" what's already there and working, but just want to try out an alternative. Branches are the way to do that in `Git`. In the `Git` supporting documents, a branch for trying something out (i.e. adding a new feature) is often referred to as the "*feature branch*."

Creating a branch is easy. In your `Git` GUI there's probably a button for doing so at the top of the interface. You can also do it within `RStudio` and on `GitHub`. To create a branch using command-line, type `$git checkout -b` *`branchname`*. `Git` will not duplicate any of your files, but it will keep track of all changes made within the branch (when you commit them to the branch). If you add, edit, or remove files (and commit) while your on your feature branch, those changes will not appear in your master branch (and vice versa). Thus, if you were to add a file while in your feature branch and were then to switch back to the master branch (using your `Git` GUI, RStudio, or `$git checkout master`[8]), you wouldn't see that new file your project directory.

The best way to think about your *master* branch is the way a software developer would: the master branch contains your clean, functioning, usable software (or as close as you are to developing it). All other branches are for trying things out. You may end up with many parallel branches, one each for testing out changes to each of several scripts or manuscript sections. Only

[6]See `Faster Computing` class later.

[7]Or you just got rejected from *journal-that-doesn't-deserve-to-publish-your-work-anyway* and want to reformat your manuscript for another journal, but want to hang on to the original because you may need to move to journal #3 and would want to start from the first journal's version to do so.

[8]Note that the `-b` used previously was only to create the new branch; you don't use it to switch to an existing branch.

once you're satisfied with your changes on the feature branch, and have made sure that they work as intended for their specific purpose and in the grand scheme of things (i.e. they don't introduce bugs or affect problems elsewhere in your code), do you bring them back into the master branch and override what was there. That's done by *merging* the feature branch into the master branch (see next section).

The other context in which branches are extremely useful is in collaborative settings where you're working on analyses or a manuscript with others. The workflow is similar to the above in that the master branch contains the not-to-be-broken best of what you've got. Each collaborator creates task-specific branches in which to work (e.g., "I'll work on (re)writing the Methods section," or "I'll work on a script to do this part of the analysis.")[9]. Then, in collaborative contexts, it's often useful to add one more step before merging: to issue a *Pull Request* (in `GitHub`) that asks the other person to review your work before implementing the merge (see below).

Now at some point you will probably end up with more than one feature branch. You might even have created feature branches off other feature branches. Or you might like to go back and look at the branching relationships and history of your repository. While `Git` has a command-line way of visualizing this history, the visualizations provided by `Git GUI`s are more informative and easier to navigate (Fig. 2). You can also use `GitHub` to see the history by going to `Insights>Network` (Fig. 3). (It can take a while for the graph to generate.)

Differencing & Merging

`diff` for highlighting changes

So now you've made changes to your code or manuscript, but before you commit those changes you'd like to compare what you've done to what you had before (in the same branch). That's easy using `RStudio` or any `Git GUI`. In `RStudio` you can see the changes by clicking on `Diff` (or `Commit`) and selecting the specific file from within the staging area. Red lines of code with minus symbols at the start are removals, green lines of code with plus symbols are additions. In `SourceTree` and most other `Git GUI`s, simply

[9]Remember to make sure your local master branch is up-to-date before creating a branch by first doing a *Pull* from `GitHub` before you create a branch.

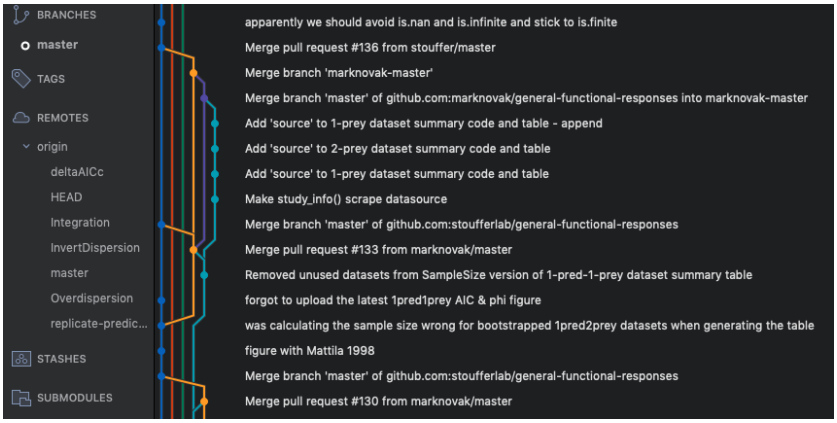


Figure 2: `SourceTree`'s branch visualization.

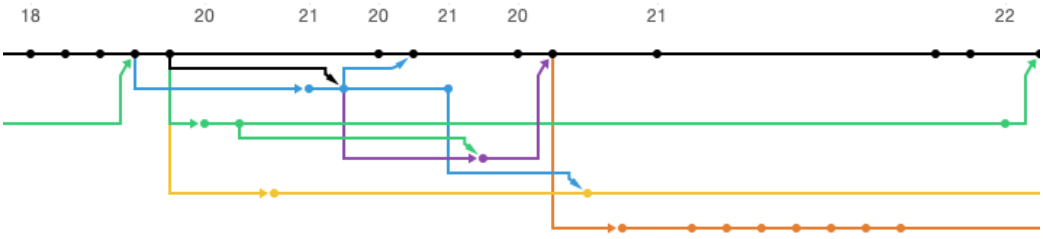


Figure 3: `GitHub`'s "Network" branch visualization.

selecting the specific file in the "unstaged" area will cause the changed lines to be shown in an adjacent window with similar highlighting.[10]

Similarly, using `Git GUIs` or command-line `Git`, you can compare the changes between your current working files and an older commit, or between any pair of commits within your branch or across branches. In `SourceTree`, simply view the *History* and select the two commits you want to compare using `Ctrl + left click`. For command-line options, see `https://git-scm.com/docs/git-diff`.

[10] When you start using LaTeX, notice that `Git` can only differentiate lines ending in a hard-return. Hence, any sentences or paragraphs that use line-wrapping won't be differentiated, causing even just a single-word edit to make `Git` think the whole sentence/paragraph has changed.

`merge` for collapsing branches

You're happy with what's in your feature branch and are ready to bring it into your master branch. In many cases (when there are no conflicts) this is exceptionally easy. In your `Git GUI`, simply switch to the branch (i.e. the master branch) into which you'd like to merge the feature branch, click on the *merge* button, select the feature branch, merge, and commit the new changes with a useful commit message. For command-line, use `$git merge` ***`feature branch`*** and then commit. Remember: If you're working collaboratively, make sure your local copy of the master branch is up-to-date by doing a *Pull* from `GitHub` before attempting to merge (as you did before creating the feature branch). That will help a lot to reduce merge conflicts[11].

Now, something to consider before you merge branches is your merging pattern. The most obvious pattern is to merge your feature branch into your master branch. That's the quickest, and introduces little risk when you're working solo or when you're adding new files that are unlikely to entail merge conflicts. The alternative to consider (especially in collaborative settings) is to merge the master branch into your feature branch, test that everthing works, then merge the updated feature branch back into the master. This pattern reduces the risk of new bugs in the master that may have be introduced by having two sets of parallel changes. It also reduces the risk of having to resolve merge conflicts on the master branch, which could affect your collaborator's ongoing efforts. The disadvantage of this second merge pattern is that it takes an extra step and results in double the number of merge commits. An alternative that solves this problem is *rebasing*, but that comes with other potential challenges.[12]

When there are no conflicts, `Git` will simply amalgamate[13] the two branches. If you're happy with the result (which you certainly should be if you've followed the second merge pattern), then you can delete the feature branch (e.g., `$git branch -d` ***`branchname`***).

[11] Just like being clear with your collaborator regarding what sections you're working on so that they don't work on the same sections (though this isn't a fundamental problem).

[12] https://git-scm.com/book/it/v2/Git-Branching-Rebasing

[13] fanciest sounding synonym for merge I can think of

Resolving merge conflicts

When `Git` notices that the same line of code has been changed on both the feature branch and the master branch subsequent to the feature branch having been created, it will tell you that there's a conflict. Either you or your collaborator may have made the change on the master; how should `Git` know which of these parallel changes should take precedence? When that happens, you'll have to resolve the conflict manually, then commit those fixes before being able to proceed.[14]

First, identify the file that contains the conflict. Your `Git GUI` should have identified the problem file(s), but you can also use `$git status` via command-line. Open the file in your text editor and search the file for the conflict marker <<<<<<<. Changes present in the HEAD (in our case master branch) will be after the line <<<<<<< `HEAD`. Next, you'll see =======, which divides these changes from the changes in the feature branch, followed by >>>>>>> ***featurebranch***. In the example below, one person edited the focal sentence in the master branch to read "feature branch into the master branch", while the other person edited the same sentence in the feature branch to read "master branch into the feature branch".

```
I prefer to merge branches by merging the
<<<<<<< HEAD
feature branch into the master branch.
=======
master branch into the feature branch.
>>>>>>> featurebranch
```

Edit the sentence the way you want it to read (you can change it entirely if you'd like), remove all the lines that have <<<<, >>>> or ==== in them (i.e. create clean text), save the file, commit the change, and you're good to go.

Pull Requests

Merging branches (as we just did) can also be done in `GitHub` by creating a *Pull Request*. When you create a pull request in `GitHub`, there's a text box for (optionally) describing the changes that you're suggesting should be merged into the master branch. Once submitted, a Pull Request initiates

[14] Admittedly, this *can sometimes* be a little annoying until you get the hang of it.

a process in which GitHub compares the two branches you selected (using diff) but then, regardless of whether there are conflicts, offers your collaborators (to whom you can assign the task of reviewing) the opportunity to add comments or request additional changes to your feature code before the merge is performed[15].

Note that you can initiate a Pull Request at any point during your coding process, even if all you want to do is share screenshots or general ideas, or when you're stuck and need an additional brain to think about the problem. Also, note that you can also continue to commit additional changes and push to the feature branch even after having initiated a Pull Request for it. GitHub will show include these additional commits in the Pull Request view, so you can go back-and-forth with your collaborators, commenting and making revisions, until you're ready to merge.

Forks vs. clones & branches

Cloning your project is what you did at the very start of this class when you created your repository on GitHub, copied the url link to it, and then provided that link to RStudio, your Git GUI, or Git itself to initiate the copying of everything that was on GitHub onto your hard-drive. You could also have gone the other way: create the Git repository on your hard-drive and then clone it to GitHub. You can work on your project from multiple computers by cloning the repository to them. (Just remember to commit and push what you're working on on the first computer in order to pull and keep working on it on your second computer.) You are the owner of that repository. Only you can pull and push to it unless you've granted permission for a collaborator to so[16]. In fact, whether public or private, you and your permission-granted collaborators are the only one that can do anything to the repository (such as create branches).

Forks are in many ways similar but are different from branches, and serve a different purpose. Anyone can fork a public repository. When you fork someone's respository you create a copy of their repository over which you have control. A reason for doing that might be wanting to contribute to someone else's project (e.g., a consortium of biologists all contributing to the

[15]Note that you can create a Pull Request and perform the merge yourself; you don't need to have a collaborator to use GitHub for merging.

[16]Go to the repository's *Settings>Manage Access* in GitHub.

maintenance of a database, or a team of R-package developers). Or it might be that you want to use someone's project (or even one of your own old projects) as a starting point for your own new project. What forking does is maintain a connection between the parent repository and the descendent repository (as opposed to simply copying files, the way you would do without `GitHub`). This permits the added convenience of being able to submit Pull Requests between forks, either to offer your improvements or additions to the parent repository, or to pull in added improvements or fixes to bugs that were identified subsequent to your having created the fork. Just as for branches, you issue a Pull Request using the "Compare and Pull Request" button, but this time you'll compare across forks rather than your internal branches.

Going back in time

We already talked about how easy it is to use `diff` to compare back to prior commits, or to compare any two commits in your repository's history. But what if you want to go back in time to revert to one of those previous commits?

Undoing commit(s)

If it's only your last commit that was made in error and you want to undo it, you have a few command-line options using `reset`[17]. To undo the last commit completely, use `git reset --hard HEAD^`. Changes that were made after the commit-before-last will be lost forever. To undo the last commit but keep your subsequently-changed files in the *unstaged* state, use `git reset HEAD^`. Changes that were made after the commit-before-last will remain but will be unstaged. To instead keep the changed files in the *staged* state, use `git reset --soft HEAD^`. To revert to any older commit you'll need its ID number. There are many additional options too, including the option to revert just a single file. (See https://git-scm.com/docs/git-reset for more.)

In `SourceTree` you can do the same things by right-clicking on the commit you'd like to revert to (not necessarily the commit-before-last) and selecting

[17]If you just want to make a small change to the last commit (e.g., you forgot to include a file or had a typo), you can `amend` the last commit without creating a new commit. Note, however, that this is changing history(!), and cannot itself be amended.

`Reset master to this commit`. For the latter you'll then be presented with three options: *hard*, *mixed*, or *soft*. *hard* will cause all changes that were made after the selected commit to be lost forever. *mixed* (the default) will keep all changed files in the *unstaged* state. *soft* will keep all changed files in the *staged* state. If it's just the last commit you'd like to undo you can also right-click on the erronous last commit and select `Reverse commit`. This won't delete the erronous commit (as the above options do) but rather will simply create a new commit equivalent to the commit-before-last. Note that if there are commits pushed to `GitHub` (by a collaborator) that you didn't first pull, those commits will "come back" the next time you pull.

One thing to recognize when you're using `GitHub`: Your last push to `GitHub` is your final recovery scenario in that you can always delete your entire local repository and re-clone (Fig. 4). Sometimes you end up messing things up enough to make that the easiest option.

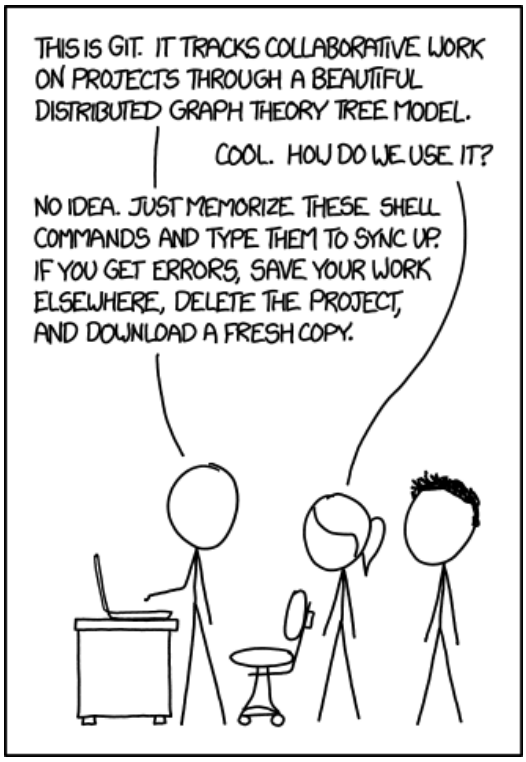


Figure 4: Solving errors in Git (source: https://xkcd.com/1597/)

Recapturing older commits

You may at some point realize that you had something in a previous commit that's worth rescuing from the past. But you may not want to discard or go through the trouble of reintegrating all the subsequent work you did, as may be necessary when using `reset`. There are a few workflows that people use to retrieve the past. For example, to retrieve specific files or directories from an old commit, use `git checkout` *`commitID filename1 dir1 dir2`* then commit, but note that this will overwrite the current copies of the files. To retrieve and reuse a whole commit, *checkout* the old commit in question, create a new *branch* off the old commit, then *merge* back into your master commit and resolve the conflicts in the process. The processes are equivalent in `SourceTree` when you right-click on the old commit or files. Notice that in the latter method lies an important reason for the `Git` motto, *commit early, commit often.* The more frequently you commit (and the better you are at commiting good "units" of change), the more targeted (and cleaner) your strikes to the past can be.

Project Management

Everyone has their own ways of managing their life, keeping track of To-Do's, etc. Not surprisingly, that's true for research projects as well. However, when you're already using `Git` to keep track of the contents of your project, its tight integration with `GitHub` enables a few very nice conveniences to keep track of both the project and its management in one place. These features are useful not only in collaborative contexts but also when working solo. We'll touch on only two of them, and only by means of a barebones introduction. Other features include the ability to host project-specific websites[18] and to create Wiki pages (which you could use as a lab notebook or to create a user manual).

Issue tracking

By now you've undoubtedly already seen `GitHub`'s Issue tracking interface. The basics are pretty simple. Create a short, specific, informative and

[18] You can even use `GitHub` to host your own personal website.

usefully-unique subject. Be concise and clear in your subsequent description of the issue. You can use `Markdown` to format your comment, and use *Preview* to make sure it looks right before posting. When your issue is related to a previously-posted issue (even a closed one), link to it using *#issueID*. When applicable, tag your issue with a label. You can add to or edit the default labels to make them more useful for specific project. By assigning issues or using *@mentions* in your comment, others will receive a notifcation (or email, if they're set up to receive them in their notification preferences). When you close an issue, it's wise to reference the commit in which the fix was enacted. You can always reopen an issue. To see all closed issues in the list of issues, simply delete the default contents of the filter search (i.e. delete *"is:open"*). If you've already setup a *Project board*, you can also assign the issue to it.

Project boards

There are a number of stand-alone apps that provide so-called Kanban boards for project management. The most basic setup entails three columns – *To do*, *In Progress*, and *Done*. When you have a new task that needs doing, you add a card to the *To do* column, then move it over to the next column as your actions on it progress.

In `GitHub` you can have multiple project boards per repository and use them for anything you'd use stand-alone Kanban app for. In my mind the motivation to use `GitHub`'s project board feature is that (1) everything to do with your project is in one place (major downside being it's only online), and that (2) they integrate nicely with Issues. Assigning an issue to a project board will cause it to appear in the "To do" column. Closing the issue will move it to the "Done" column You can set up your project board so that any newly-created issues are automatically added to the "To do" column.

Workflows for Visualization

Introduction

Visual analogies often play a key role in research, helping us articulate hypotheses and understand results of hypothesis tests. Yet, while hypothesis testing is a core component of science education, visualization, and, more generally, the acts of perception and imagination that necessarily precede hypothesis testing, do not receive as much attention in our curricula, despite their critical importance.

For one thing, the basics of making good figures, which are explicitly known and practiced by most successful scientists, are rarely formally taught. Indeed, I believe the association between knowing how to make a good figure and being a successful scientist is causal.

More generally, as biologists we are often working with complex systems. Visualizing complex systems requires us to bring into focus specific emergent properties of a system, by mapping to a simpler context, and by leaving most things out. In this sense, visualizing complex systems is similar to building mathematical models. Indeed visualizing complex systems requires some mathematics, and often catalyzes further quantitative analyses.

Here's a cute argument: a huge component of our brains is evolved for spatial reasoning - for instance, to help us find food and not become food for another creature. Think of this like the GPC (graphics processing card) of your brain-computer. Your internal GPC is so much more powerful than your abstract processing abilities (the CPU of your brain-computer). We can do better science by working better with our GPCs.

Example of a basic visualization workflow

Here we chronicle the journey from a simple dataset to an acceptable figure, in base R.

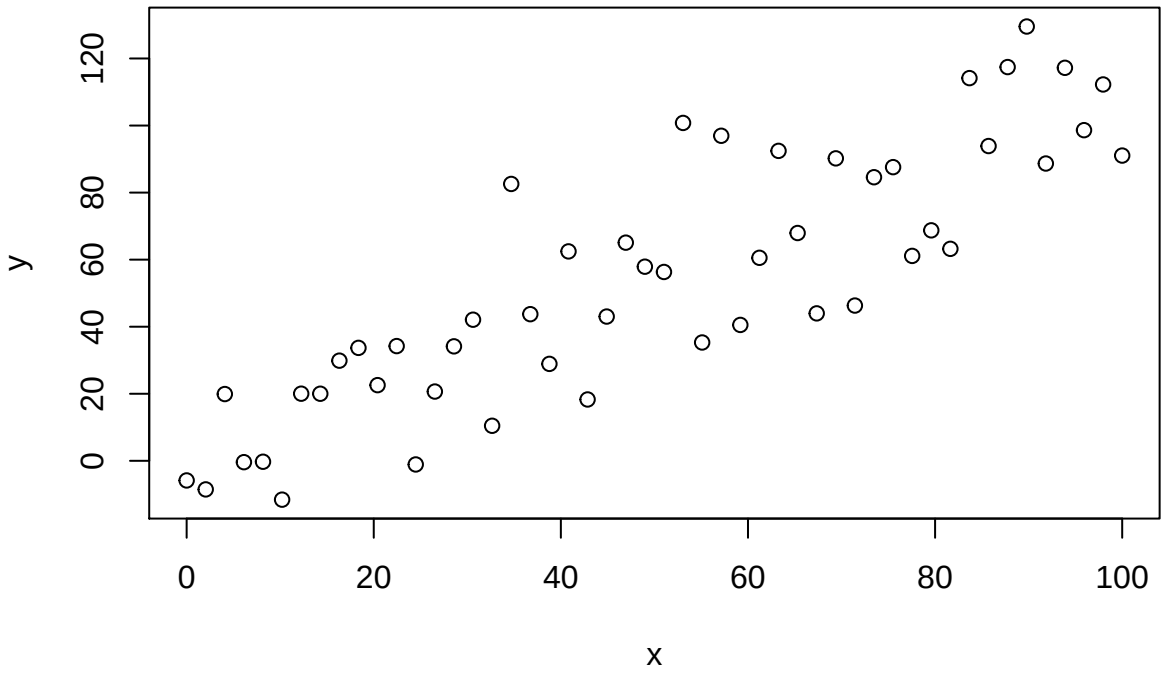
Setup

We will start by generating some data, as an additive combination of a linear "signal" and some white noise

```
n <- 50
a <- 1
b <- 2
sig <- 20
noise <- rnorm(n, 0, sig)

x <- seq(0, 100, length.out = n)
signal <- a * x + b
y <- signal + noise
```

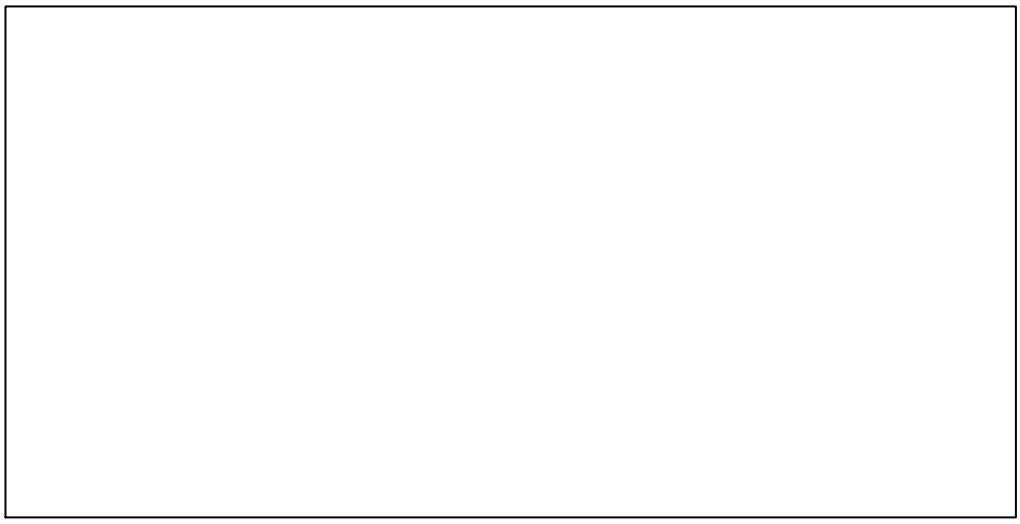
At this point it is worth spending some time in the help files for `plot()` and `par()` if you have not already. This is just to get a sense of the potential to control things, and how the basic control process works. Most people need to continually refer to these help files as they work on specific applications.

```
plot(x, y)
```



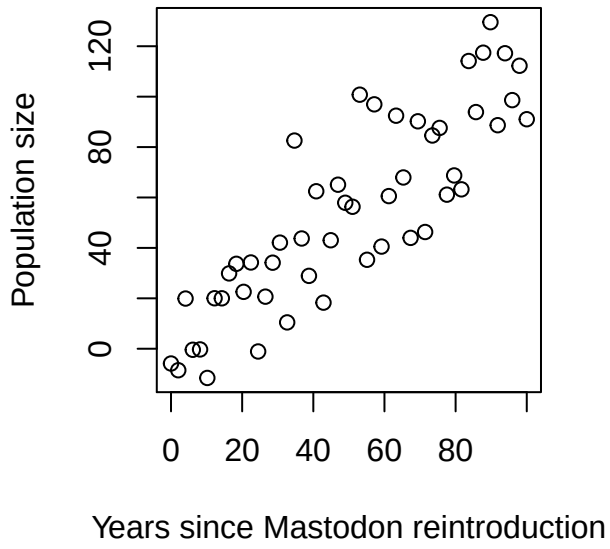
Starting a plot from the VERY beginning - (i.e. all defaul elements turned off) - if you want to control everything manually.

```
plot(x, y,
     type = "n", # do not even plot the data
     #bty = "n",  # box type is "n" means don't draw a box around the plot
     xaxt = "n", # similarly, don't make an x axis...
     yaxt= "n",  #"... or a y axis"
     xlab = "",  # and axis labels are blank
     ylab = ""
     )
```



Before tweaking optional things, there are some minimum standards to take care of. Every figure should have informative axis labels. And these need to be large enough to be read when the figure appears in a published article. To this end, it can be helpful to control the size of a figure right from the start. If you have not already, spend some time in the help files for `plot()` and `par()`.

```
par(fin = c(4,4))           # figure dimensions (width, height)
par(mai = c(1,1,1,1))       # plot margins (bottom,left,top,right)
plot(x, y,
     xlab = "Years since Mastodon reintroduction",
     ylab = "Population size")
```



By labelling the axes I have just realized there are y values that make no sense given what y is: population size. This is a simplistic example of how disciplined practices in making figures (e.g. always having informative axis labels from the start) can improve the whole scientific process: now we have an opportunity to improve our data stream by figuring out what to do with our negative y values. For our purposes here, let's just remove them:

```
y[y < 0] <- NA
```

Now lets start work on minimizing the amount of effort a reader's brain has to do to "get" the figure. There are some initial things that apply to every figure that we can do right away. Then in the next section we can look at making specific messages come forward.

Little things can go a long way. Custom tick marks are usually needed to make professional-grade figures. To do that, we need just enough tick labels to orient the reader to the "space" of the plot and help them quickly measure distances; any tickmarks beyond that just become visual noise the reader has to work to filter out. We start by turning off the default axes. Then we add our own, with axis limits hardcoded.

From a reproducibility standpoint, hard-coding is a liability. The alternative is to write code that determines what the best limits should be on the fly, but we will not be distracted by that now.

```
par(fin = c(4,4))
par(mai = c(1,1,1,1))
plot(x, y,
     xlab = "Years since Mastodon reintroduction",
     ylab = "Population size",
     xaxt = "n",
     yaxt = "n",
     xlim = c(0, 100),     # hardcoded axis limits...
```

```
     ylim = c(0, 120)
     )
axis(1, c(0, 50, 100))    # ... and tick locations
axis(2, c(0, 60, 130))
```

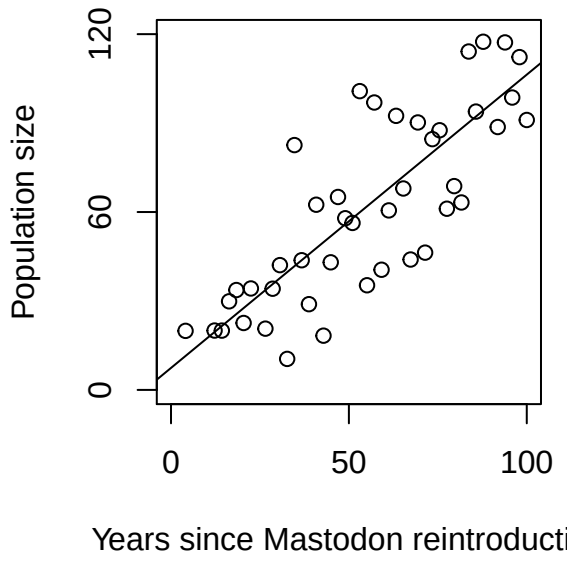
Tune for impact

Now that we have something that meets minumum formating requirements we can ask: what are we trying to say with this figure and how can we make that pop? Let us say our main message here is that the Mastodon population is increasing in a predictable way. That would be the first line of a figure legend: "i.e. Figure 1: Following reintroduciton in 2023, the Mastodon population has increased in a predictable way." This is also the first sentence you would say in a presentation of this slide. The task at hand is to make the visual say that loud and clear. We need a regression line.

```
fit <- lm(y ~ x)
```

We could add this to the figure a number of ways, for example, using the function `abline()` and supplying the fitted model as input.

```
par(fin = c(4,4))
par(mai = c(1,1,1,1))
plot(x, y,
     xlab = "Years since Mastodon reintroduction",
     ylab = "Population size",
     xaxt = "n",
     yaxt = "n",
     xlim = c(0, 100),
     ylim = c(0, 120)
     )
axis(1, c(0, 50, 100))
axis(2, c(0, 60, 120))

abline(fit) # add linear regression line to an existing plot
```



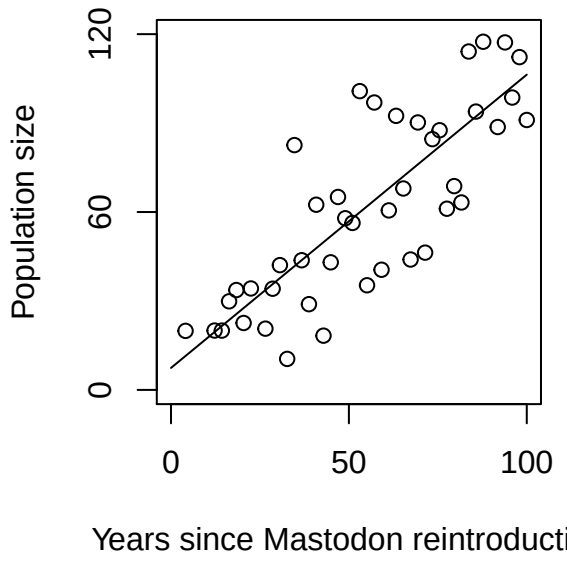
We get a lot more control of how the predictions are displayed if we generate an additional dataset that is the prediciton, then plot that. For example, using `abline()` causes the fitted line to extend beyond the data, which is not only statistically bad (we are typically interpolating, not extrapolating when fitting a linear regression model), but also doesn't make biological or physical sense (we can't go to negative time since the mastadon reintroduction).

First generate the predictions:

```
npred <- 10
xpred <- seq(min(x), # points at which we want to predict
             max(x),
             length.out = npred)

ypred <- predict(fit, # predicting corresponding y values and se
                 newdata = data.frame(x = xpred),
                 se.fit = TRUE)
```

Then plot:

```
par(fin = c(4,4))
par(mai = c(1,1,1,1))
plot(x, y,
     xlab = "Years since Mastodon reintroduction",
     ylab = "Population size",
     xaxt = "n",
     yaxt = "n",
     xlim = c(0, 100),
     ylim = c(0, 120)
     )
axis(1, c(0, 50, 100))
axis(2, c(0, 60, 120))

lines(xpred, ypred$fit)
```



This also allows us to make the plot more informative and visually appealing:

```r
# Original code, modified to not plot the points at first
par(fin = c(4,4))
par(mai = c(1,1,1,1))
plot(x, y,
     type = "n",                                    #first plot with points missing
     xlab = "Years since Mastodon reintroduction",
     ylab = "Population size",
     xaxt = "n",
     yaxt = "n",
     xlim = c(0, 100),
     ylim = c(0, 120)
     )
axis(1, c(0, 50, 100))
axis(2, c(0, 60, 120))

# Draw a polygon for the confidence envelope
xpoly <- c(xpred, rev(xpred))                       #add confidence envelope as bottom layer
ypoly <- c(ypred$fit + 5 * ypred$se.fit,
           rev(ypred$fit - 5 * ypred$se.fit))

polygon(x = xpoly, y = ypoly,
        border = NA,
        col = grey(0.8))

# Add in the points and line
points(x,y)                                         #then data
lines(xpred, ypred$fit, col = 'white')              #then trendline
```

That takes care of our primary messsage: "Mastodon population is increasing in a predictably-linear way." What about secondary messages? Suppose we are interested in what caused the population to grow especially quickly or slowly in some years. We can use symbol color and shape to highlight points that are outside the confidence envelope.

To do that we need to have the model prediction and standard error for each observation point, rather than at 10 evenly spaced points as we had before.

```
yhat <- predict(fit, newdata = data.frame(x = x), se.fit = T)
```

Now identify points that are above (boom years) and below (bust years) the confidence envelope.

```
is_boom_year <- y > yhat$fit + 5*yhat$se.fit
is_bust_year <- y < yhat$fit - 5*yhat$se.fit
```

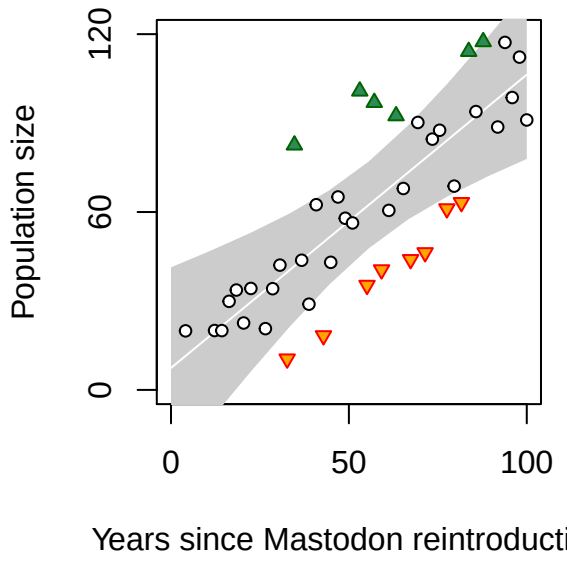
Use symbology to reinforce the message. To do that we first specify size, shape and color as vectors, mapping from data. Then we call the plot function. It's generally wiser to do this outside rather inside the plot function call because the latter is harder to read and debug.

```
# Default vectors for symbology
cex <- rep(1, n)        # size
col <- rep("black", n)  # color
pch <- rep(21, n)       # shape.  See ?points


# Change to map non-default symbology to data
cex[is_boom_year || is_bust_year] <- 2
col[is_boom_year] <- "green"
col[is_bust_year] <- "red"
pch[is_boom_year] <- 2
pch[is_bust_year] <- 6
```

The most important thing is that it pops: do not rely on subtle differences in color or shape to get across key results. One way to achieve that is to simultaneously alter multiple features, such as changing shape, color and size together (but be careful not to overdo this and create confusion when working with multiple legends, and be sure to remain consistent across figures).

```
# Vectors for symbology
cex <- rep(0.8, n)      # size
```

```r
col <- rep("black", n)  # color
bg <- rep("white", n)   # background color
pch <- rep(21, n)       # shape.  See ?points


# Change symbology to highligh features in data
cex[is_boom_year || is_bust_year] <- 1.3
bg[is_boom_year] <- "seagreen"
bg[is_bust_year] <- "orange"
col[is_boom_year] <- "darkgreen"
col[is_bust_year] <- "red"
pch[is_boom_year] <- 24
pch[is_bust_year] <- 25


# Make plot first without points
par(fin = c(4,4))
par(mai = c(1,1,1,1))
plot(x, y,
     type = "n",
     xlab = "Years since Mastodon reintroduction",
     ylab = "Population size",
     xaxt = "n",
     yaxt = "n",
     xlim = c(0, 100),
     ylim = c(0, 120)
     )
axis(1, c(0, 50, 100))
axis(2, c(0, 60, 120))


# Draw a polygon for the confidence envelope as bottom layer
xpoly <- c(xpred, rev(xpred))
ypoly <- c(ypred$fit + 5 * ypred$se.fit,
           rev(ypred$fit - 5 * ypred$se.fit))

polygon(x = xpoly, y = ypoly,
        border = NA,
        col = grey(0.8))


# Add trendline
lines(xpred, ypred$fit, col = 'white')


# Add in the points, with their symbologies pre-specified above (no calculations here)
points(x,y,
       pch = pch,
       col = col,
       bg = bg,
       cex = cex)
```



Now this figures says:

1. "Mastodon population is increasing in a predictably linear way."
2. "We are going to focus on the years where growth was particularly fast or slow relative to those predictions."

Other graphs in your paper should use the same color scheme and symbology to denote analyses pertaining to the boom and bust years.

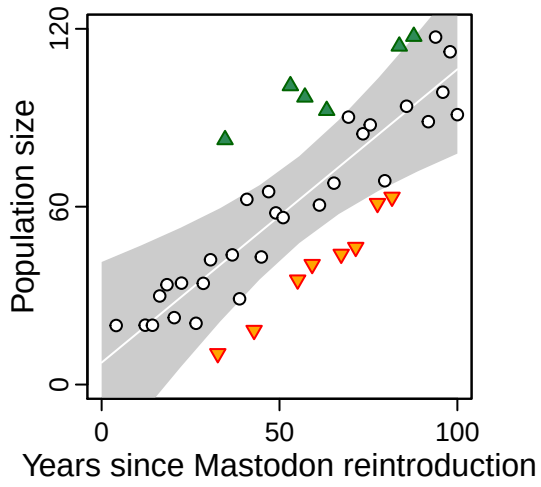
Final touches

There are still a few things that might make this figure just a little more visually appealing:

1. The length of the axis tickmarks.
2. The size of the axis labels relative to the axis titles.
3. The space between the axis tickmarks and the axis labels
4. The space between the axis titles and the box.
5. A solid box (that doesn't have the grey of the confidence overlapping it).

```
# Vectors for symbology
cex <- rep(0.8, n)      # size
col <- rep("black", n)  # color
bg <- rep("white", n)   # background color
pch <- rep(21, n)       # shape.  See ?points


# Change symbology to highligh features in data
cex[is_boom_year || is_bust_year] <- 1.3
bg[is_boom_year] <- "seagreen"
bg[is_bust_year] <- "orange"
col[is_boom_year] <- "darkgreen"
col[is_bust_year] <- "red"
pch[is_boom_year] <- 24
pch[is_bust_year] <- 25


# Make plot first without points
```

```r
par(fin = c(4,4),
    mai = c(1,1,1,1),
    tcl = -0.3,      # length of tickmarks
    mgp = c(1.2, 0.3, 0),
    cex.axis = 0.8) # distance of axis title, axis label, and axis line to box
plot(x, y,
     type = "n",
     xlab = "Years since Mastodon reintroduction",
     ylab = "Population size",
     xaxt = "n",
     yaxt = "n",
     xlim = c(0, 100),
     ylim = c(0, 120)
     )
axis(1, c(0, 50, 100))
axis(2, c(0, 60, 120))

# Draw a polygon for the confidence envelope as bottom layer
xpoly <- c(xpred, rev(xpred))
ypoly <- c(ypred$fit + 5 * ypred$se.fit,
           rev(ypred$fit - 5 * ypred$se.fit))

polygon(x = xpoly, y = ypoly,
        border = NA,
        col = grey(0.8))


# Add trendline
lines(xpred, ypred$fit, col = 'white')


# Add in the points, with their symbologies pre-specified above (no calculations here)
points(x,y,
       pch = pch,
       col = col,
       bg = bg,
       cex = cex)

box(lwd = 1) # place a box over the plot
```



Remark

A corrolary of this approach is that figures should have a small number of clear, concrete messages. For any figure you show your audience, you should be able to explicitely populate a (short) list like the one above ("This figure says:"). Having done that, you can evaluate how efficiently the figure delivers its message. A figure does not have to show every aspect of the data, or even most aspects. Every display element should be critical to the core message. In other words, if you can leave it out with compromising the core message, you should do so. If you are worried you are misleading readers by leaving something out, include a note in the figure legend about what was omitted, and consider including a more complete (less efficient) version of the figure in the supplemental materials.

This will be my awesome title.

An Intro to LaTeX

November 10, 2020

Contents

1 Introduction **2**

2 Basics **2**

2.1 Document structure 2

2.2 Basic equations 2

2.3 Lists 3

2.4 Tables 3

2.5 Within-text referencing 3

3 More equations **4**

4 Inserting figures **4**

5 Inserting external tables **5**

6 References **5**

1 Introduction

I think it's fair to say that most biologists switch from using word processors (like Microsoft Word or Apple Pages) to LaTeX when they start writing papers containing more than just one or two equations. I have to admit though, that after using both LaTeX and Pages for a few years (depending on the type of manuscript I was writing), I've now switched entirely to writing manuscripts in LaTeX (when collaborators allow). It took me a while to get the hang of it, but now I prefer it even when there are no equations involved.

Note that (almost) all the lecture notes for this class were written in LaTeX so take a look at a few of their `tex` files to learn a few additional tricks.

Note that your LaTeX file is code that gets compiled.

2 Basics

2.1 Document structure

What isn't visible here in this pdf document is that the code for every LaTeX document begins with a preamble that sets things up. In the preamble you define the type of document it is, load necessary packages, define any additional functions you might like to have, and provide your author name and the title of the paper. Your actual text is then written after beginning your document using `\begin{document}` and is followed at the very end by `\end{document}`.[1]

You "comment out" lines with the % symbol. Also, if you look at the raw `.tex` file behind this pdf, you'll notice that I wrote each sentence on its own line (i.e. with hard-returns between each). That's not necessary, but allows `Git` to distinguish changes on a sentence-specific rather than whole paragraph basis. A common alternative is to setup your editor such that all lines are limited to 72 or 80 characters.

2.2 Basic equations

The fundamental principle of calculus entails

$$\lim_{\Delta t \to 0} \frac{f(a + \Delta t) - f(a)}{\Delta t}. \tag{1}$$

[1] Many other functions use the same begin ... end structure.

Although the typical way of writing a derivative is $\frac{dx}{dt}$, some fields also write it as $\dot{x}$.

2.3 Lists

Calculus rocks because it can be used to represent all of the following:

- first item
- second item

It also rocks because

1. first item
2. second item

2.4 Tables

One could also organize all the things calculus can be used for in a table:

Reason	Explanation
1	blah blah blah
2	blah blah blah

2.5 Within-text referencing

I can easily reference the section (sect.1), equation (eqn. 1) and table (Table 2.4). Their numbers will auto-generate, which makes it easy to move them around in your paper and adhere to a journal's stylistic preferences.

My un-numbered subsection

Sections and subsection are numbered by default, but that can be overwritten for a given section, or globally using `\setcounter{secnumdepth}{0}` in the preamble.

Figure 1: This is the LaTeX logo.

3 More equations

Using align

The Lotka-Voltera equations are given by

$$\frac{dx}{dt} = \alpha x - \beta xy \tag{2}$$

$$\frac{dy}{dt} = \gamma xy - \delta y \tag{3}$$

Often it's useful to typeset the steps of derivations. Pay it forward to folks who are trying to learn these methods, and to yourself when you can't remember the details but have to lecture on it in 5 minutes. In these cases you don't need to number each line.

$$\begin{aligned} N(10) &= \lambda N(9) \\ &= \lambda^2 N(8) \\ &= \lambda^3 N(7) \\ &\dots \\ &= \lambda^{10} N(0) \end{aligned} \tag{4}$$

4 Inserting figures

Note that the position of figures is auto-determined (e.g., Fig. 1)! You can force the position of figures using the `float` package and then the [H] option for your figure (e.g., Fig. 2).

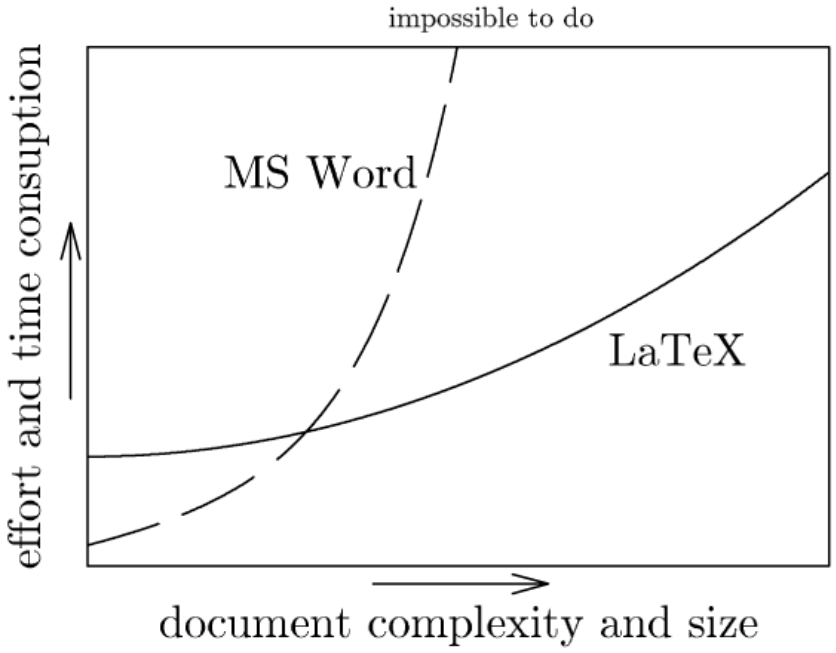


Figure 2: Why use LaTeX (source: http://www.pinteric.com/miktex.html)

5 Inserting external tables

It's relatively easy to use the R `Hmisc` package to generate LaTeX tables (see `../R/ExportTable.R`) and then import them into your document using `input`. Again, just like figures, their placement in the document is auto-determined. If you provided a caption to the tables when generating their tex files in R, it's easy to reference them (Table 1 and 2).

6 References

The `natbib` package is great for citing references. Reformatting for a different journal is as easy as changing the arguments of a function. How to cite references and include them in a bibliography is demonstrated in the accompanying `manuscript.tex` template in the parent folder of these lecture notes.

Table 1: The first six lines of my dataset.

x	y
0.443263064604253	4.28083287471942
0.892026401590556	5.13777481203160
0.509197412524372	3.67473555434176
0.967874688329175	5.37934574007462
0.810071667889133	4.94420398679803
0.485601465217769	3.81008116598662

Table 2: Estimated coefficients of a linear fit to the data.

Parameter	Estimate	Std. Error	t value	p-value
a	1.159	0.476	2.436	0.025
b	2.132	0.846	2.520	0.021

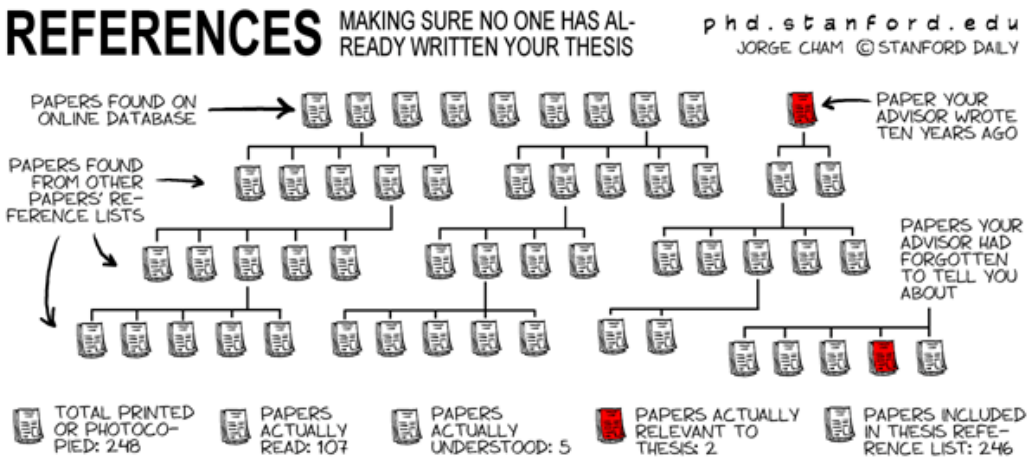


Figure 3: Reading is fundamental (source: http://phdcomics.com/comics/archive.php?comicid=286)

Faster Computing - Part 1

November 17, 2020

Contents

Overview **1**

Benchmarking **1**
Benchmarking using `Sys.time()` . . . 2
Benchmarking using `system.time()` . . . 3
Using the `microbenchmark` package . . . 3

Faster `for`-loops **4**
Preallocated space . . . 4
Progress bar . . . 5

Vectorize it! **6**
Thinking in vectors . . . 6
`apply()`-family functions . . . 6

Parallelize it! **9**
Running `lapply` in parallel . . . 9
Running `for`-loops in parallel . . . 9

Profiling **10**

Overview

Sometimes it's nice to have code that takes a long time to run; it feels like you're working even when you're not! But most of the time it's really useful when you can speed up code so that it takes seconds intead of minutes, or minutes instead of hours, or hours instead of days. Most of the time faster computing can be achieved by creativity. It's therefore wise to take the time to think of how you might achieve the same goal in a different manner, and to search online (e.g., https://stackexchange.com) for similar problems to see if others have solutions or inspirational ideas. If what you're trying to achieve requires a large number of steps, search CRAN (https://cran.r-project.org) to see if someone has created a package with useful functions. These have likely been optimized for efficiency (e.g., by performing computations in `C++` that are "wrapped" in `R` code). That said, there are many other ways to achieve faster code. Today we'll step through a few options, moving from the simplest forms of code efficiency to the use of computer clusters. Note that there are a number of additional ways for computing faster that we will not discuss (e.g., using `Rcpp` to compile your own `C++` code)

Benchmarking

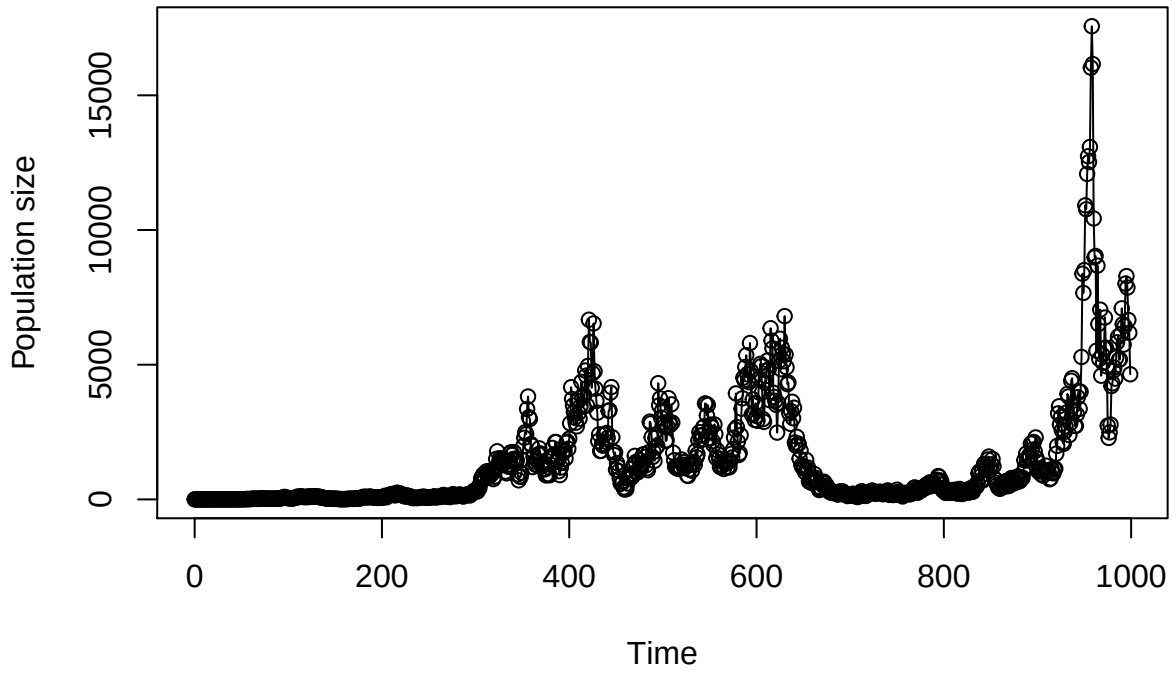
The first useful tool for optimizing code is a way to quantitatively measure and compare code performance. That's what benchmarking refers to. There's a few ways to do it, but here are examples of two simple methods using functions that are built in to base-`R`, and a third example using the `microbenchmark` package.

For demonstration purposes, let's consider the task of performing a stochastic simulation of a population that is, on average, growing geometrically. That is, given a mean growth rate $\lambda = 1.01$ whose year-to-year variation is described by a normal distribution with standard deviation $\sigma^2 = 0.2$, we wish to simulate

$$N_{t+1} = N_t(\lambda e^{\epsilon}) \tag{1}$$
$$\epsilon \sim \mathcal{N}(0, \sigma^2) \tag{2}$$

for a total of $T = 999$ time-steps starting from an initial size population size $N_0 = 2$. We'll use a `for`-loop to do it and, for convencience, will wrap it into a function with the desired parameter values as defaults.

```
set.seed(1) # for reproducibility
geom_growth_base <- function(N0 = 2,
                             T = 999,
                             lambda = 1.01,
                             sigma = 0.2){
  Nvals <- vector('numeric') # initiate a place to put the values
  Nvals[1] <- N0
  for (t in 1:T){
    Nvals[t+1] <- Nvals[t]*(lambda*exp(rnorm(1,0,sigma)))
  }
  return(Nvals)
}

# Run the simulation
out <- geom_growth_base()
# Plot the results
plot(0:999,
     out,
     xlab='Time',
     ylab='Population size',
     type='o')
```



Eerily similar to the dynamics of COVID, huh.

Now let's quantify how long it takes our function to run.

Benchmarking using `Sys.time()`

The simplest way is to ask `R` what time it is before and after running our function using `Sys.time()`.

```
# Default number of time-points
start_time <- Sys.time()
  out <- geom_growth_base()
end_time <- Sys.time()

end_time - start_time
```

```
## Time difference of 0.05011106 secs
```

```
# Repeat with greater number of time-points
start_time <- Sys.time()
  out <- geom_growth_base(T=9E5)
end_time <- Sys.time()

end_time - start_time
```

```
## Time difference of 2.825775 secs
```

```
# Note that the time won't be exactly the same each time (unless the seed is the same)
start_time <- Sys.time()
  out <- geom_growth_base(T=9E5)
end_time <- Sys.time()

end_time - start_time
```

```
## Time difference of 2.672357 secs
```

Using `Sys.time()` makes it easy to measure the run-time of any section of code, no matter how long it is, because you don't have to wrap the code in anything.

Benchmarking using `system.time()`

The function `system.time()` lets you evaluate the run-time of any expression (function), and provides two additional ways of counting time.

```
system.time(geom_growth_base(T=9E5))
```

```
##    user  system elapsed
##   2.686   0.163   2.918
```

`user` gives the CPU time spent by the R session (i.e. the current process). `system` gives the CPU time spent by the kernel (the operating system) on behalf of the R session. The kernel CPU time will include time spent opening files, doing input or output, starting other processes, etc. (i.e. operations involving resources shared with other system processes). `elapsed` gives the time as we measured using `Sys.time()`.

Using the `microbenchmark` package

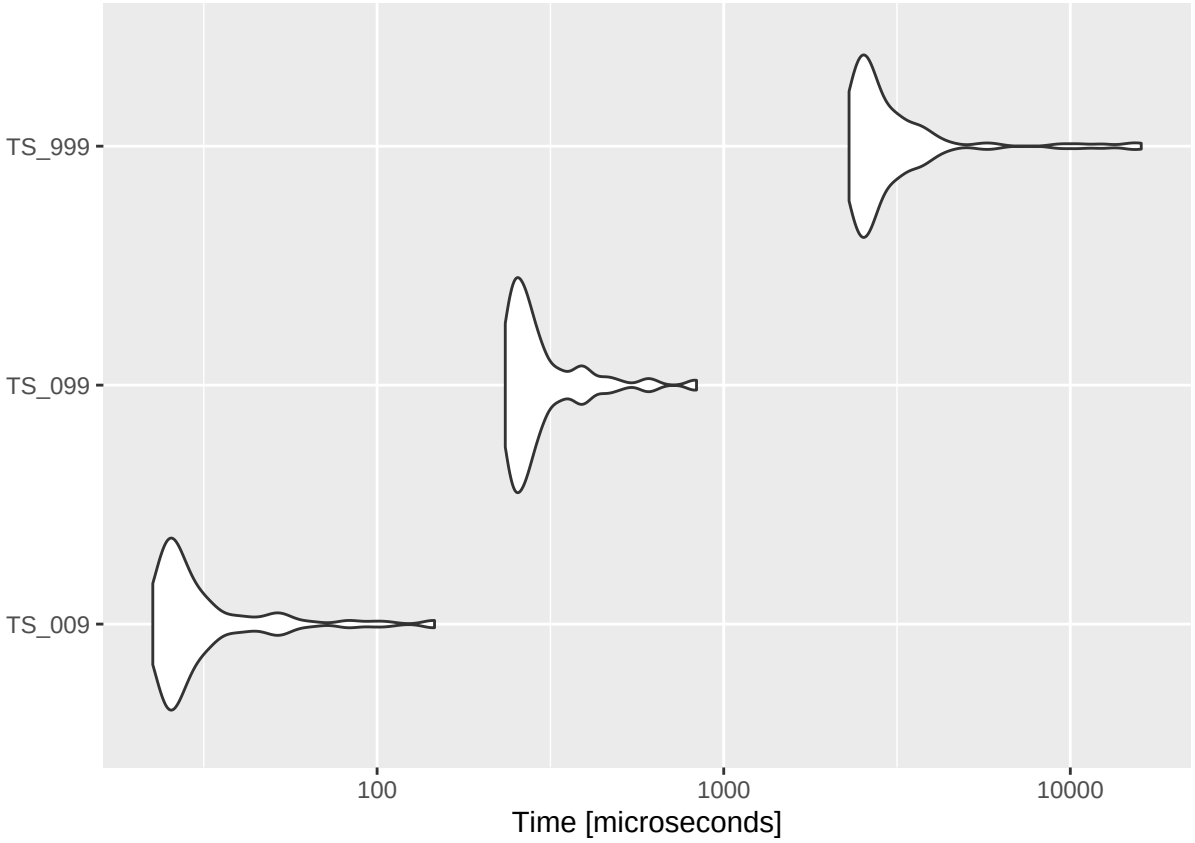
There are a few benchmarking packages out there (including `tictoc` and `rbenchmark`). They're similar in that they simplify the task of comparing the speed of multiple functions. `microbenchmark` is nice 'cuz it will evaluate each function repeatedly (default `neval`=100 times) and return summary statistics. It also plays well with `ggplot` to enable quick visual comparisons.

```
library(microbenchmark)

comp <- microbenchmark(TS_009 = {geom_growth_base(T=9)},
                       TS_099 = {geom_growth_base(T=99)},
                       TS_999 = {geom_growth_base(T=999)})
comp
```

```
## Unit: microseconds
##    expr      min        lq      mean    median       uq       max neval
##  TS_009   22.536   24.8635   34.8809   26.8650   32.577   146.450   100
##  TS_099  234.307  250.1140  308.7668  269.5485  306.757   834.773   100
##  TS_999 2299.569 2488.7315 3368.6340 2678.4085 3188.267 16019.667   100
```

```
library(ggplot2)
autoplot(comp)
```

```
## Coordinate system already present. Adding new coordinate system, which will replace the existing one
```



Faster `for`-loops

Preallocated space

Now let's start writing faster code! You may be surprised to learn that even `for`-loops can be made faster with just a tiny adjustment: pre-specifying the length of the container that will hold the results. Let's do that and compare to our old `for`-loop simulation.

```
set.seed(1) # for reproducibility
geom_growth_preallocated <- function(N0 = 2,
                                     T = 999,
                                     lambda = 1.01,
                                     sigma = 0.2){
  Nvals <- vector('numeric', length = T+1) # here's the only change
  Nvals[1] <- N0
  for (i in 1:T){
    Nvals[i+1] <- Nvals[i]*(lambda*exp(rnorm(1,0,sigma)))
  }
  return(Nvals)
}

# Compare the old and new simulation functions
comp <- microbenchmark(Old = {geom_growth_base(T=9999)},
                       New = {geom_growth_preallocated(T=9999)})
comp
```

```
## Unit: milliseconds
##  expr      min       lq     mean   median       uq      max neval
##   Old 22.11822 23.96159 28.35793 29.31266 31.29064 48.61277   100
##   New 19.27235 21.90375 27.91718 27.18749 30.13530 87.71013   100
```

Preallocating the size of containers – whether they're vectors, arrays, or lists – can make a big difference when you're using a `for`-loop – or any function – repeatedly.

Progress bar

Next we'll learn how to avoid `for`-loops altogether using vectorization, which is typically much faster. But oftentimes, slow `for`-loops are impossible to avoid given the nature of the problem (like in our population simulation example). In such cases it's often nice to know how far along your code is in its computation. You could, of course, simply include a `print()` statement just before the end of the `for`-loop (e.g., `print(paste(t, "of", T, "completed"))`), but that will quickly eat up console space unless you also use `flush.console()`. Alternatively, there are a few options for creating progress bars. The simplest that does all you really need comes with base-`R`:

```
total <- 10
pb <- txtProgressBar(min = 0, max = total, style = 3)
```

```
##
  |
  |                                                                      |   0%
```

```
for(i in 1:total){
   Sys.sleep(0.1)
   setTxtProgressBar(pb, i) # update progress bar
}
```

```
##
  |
  |=======                                                               |  10%
  |
  |==============                                                        |  20%
  |
  |=====================                                                 |  30%
  |
  |============================                                          |  40%
  |
  |===================================                                   |  50%
  |
  |==========================================                            |  60%
  |
  |=================================================                     |  70%
  |
  |========================================================              |  80%
  |
  |===============================================================       |  90%
  |
  |======================================================================| 100%
```

```
close(pb)
```

(Note that it's only because of `knitR` that each iteration of the progress bar is printed. It'll remain on a single line in the console.)

Vectorize it!

As already mentioned, `for`-loops are slow. Learning to avoid them is your best ticket to writing faster (and more compact) code. It takes practice though, so we'll start with a simple example.

Thinking in vectors

Let's say we had population dynamics data, like we simulated above, from which we want to calculate the population's growth rate between each pair of successive years.
Using a `for`-loop, we would do:

```
data <- geom_growth_preallocated(T=99999)

start_time <- Sys.time()
  growth_rates <- vector('numeric', (length(data)-1))
  for(i in 1:(length(data)-1)){
    growth_rates[i] <- data[i+1] / data[i]
  }
end_time <- Sys.time()
end_time-start_time
```

```
## Time difference of 0.01814699 secs
```

But what are we actually doing here? Instead of looping through all the data points, we could do the same by considering them as comprising nothing more than a vector. Because of the way `R` treats vectors, we could take all data points excepting the first, and divide them by all data points excepting the last:

```
start_time <- Sys.time()
  growth_rates <- data[-1] / data[-length(data)]
end_time <- Sys.time()
end_time-start_time
```

```
## Time difference of 0.004589081 secs
```

We've improved the speed of our code by an order of magnitude! Granted, writing `for`-loops is a good way to figure out what you want to do. But quite often it's good to consider them more like pseudo-code that helps you think through how to write faster functions using vector-based operations.

`apply()`-family functions

Base-R has a number of functions for vector-based operations, and there are many more in various packages (e.g., in the TidyVerse). We'll touch on the `apply()`-family here (which includes `lapply`, `sapply`, `mapply` and more). The key to successful vector-based operations often lies in figuring out how to write the function in order to "apply" it.

We'll demonstrate by continuing on with our population dynamics example. First, let's create a dataset that contains multiple population time-series.

```
n <- 5 # number of time-series to create
# use replicate() to create n time-series, each in a different matrix column
dat_array <- replicate(n, geom_growth_preallocated(T=9999))
colnames(dat_array) <- paste0('Site_', 1:n)
head(dat_array)
```

```
##        Site_1   Site_2    Site_3   Site_4   Site_5
## [1,] 2.000000 2.000000 2.0000000 2.000000 2.000000
## [2,] 2.567805 1.979162 1.9014539 1.630320 1.828893
## [3,] 4.654549 2.498457 1.5693809 1.355233 1.896816
## [4,] 3.911162 2.072130 1.2510461 2.033924 2.061827
## [5,] 2.912498 2.697578 1.4261211 1.550670 2.381372
## [6,] 3.638224 3.132035 0.9490439 1.654280 2.422750
```

Now let's define a vector-based function to calculate growth rates between all time-steps in a time-series, and apply it to each site (i.e. each column):

```
calc_growth_rates <- function(x){
  gr <- x[-1] / x[-length(x)]
  return(gr)
}
system.time({
  apply(dat_array, 2, calc_growth_rates)
})
```

```
##    user  system elapsed
##   0.006   0.000   0.007
```

```
# margin = 2 means apply to each column
# margin = 1 means apply to each row
```

In order compare what we just did to the use of `lapply`, we'll first convert the data that's currently in an `array` format (i.e. a matrix) into a `list` format. That is, we'll take each column (i.e. each population time series) and place it in its own list element.

```
dat_list <- as.list(as.data.frame(dat_array))
```

```
system.time({
  growth_rates <- lapply(dat_list, calc_growth_rates)
})
```

```
##    user  system elapsed
##   0.002   0.001   0.001
```

```
lapply(growth_rates, head) # look only at head of each list element
```

```
## $Site_1
## [1] 1.2839027 1.8126563 0.8402882 0.7446631 1.2491765 0.9920725
##
## $Site_2
## [1] 0.9895810 1.2623812 0.8293638 1.3018387 1.1610544 0.9992152
##
## $Site_3
## [1] 0.9507270 0.8253584 0.7971590 1.1399429 0.6654722 0.8227774
##
## $Site_4
## [1] 0.8151599 0.8312685 1.5007921 0.7624031 1.0668164 0.7457889
##
## $Site_5
## [1] 0.9144467 1.0371385 1.0869936 1.1549815 1.0173760 0.9308679
```

The nice thing about lists (as opposed to matrices and arrays) is that list elements need not be of the same length (e.g., you could have time-series for different populations that differ in their number of time points). Since `lapply` returns a list, performing additional computations by site (e.g., calculating an arithmetic mean growth rate) is super fast and easy. (No looping required!)

```
growth_rate_means <- lapply( lapply(dat_list, calc_growth_rates), mean)
growth_rate_means
```

```
## $Site_1
## [1] 1.029277
##
## $Site_2
## [1] 1.031072
##
## $Site_3
## [1] 1.031549
##
## $Site_4
## [1] 1.029922
##
## $Site_5
## [1] 1.033189
```

```
unlist(growth_rate_means)
```

```
##   Site_1   Site_2   Site_3   Site_4   Site_5
## 1.029277 1.031072 1.031549 1.029922 1.033189
```

The function `sapply` is similar to `lapply` but returns a vector, matrix, or array instead of a list.

```
sapply( lapply(dat_list, calc_growth_rates), mean)
```

```
##   Site_1   Site_2   Site_3   Site_4   Site_5
## 1.029277 1.031072 1.031549 1.029922 1.033189
```

The function `mapply` is useful when you want to parameterize a function from multiple vectors. For example, suppose you have a function that has two parameters and you want to run it through a series of parameter combinations:

```
dat_list <- mapply(geom_growth_preallocated,
           N0 =c(Site1 = 1.8, Site2 = 2.1), # initiate simul. with diff. N0's
           T = c(Site1= 4, Site2 = 9)) # initiate simul. with diff. T's
dat_list
```

```
## $Site1
## [1] 1.800000 1.377143 1.134109 1.449216 1.120348
##
## $Site2
##  [1] 2.100000 1.733737 1.753519 1.954439 2.040831 2.233761 2.628415 2.584526
##  [9] 3.030549 2.944133
```

```
sapply( lapply(dat_list, calc_growth_rates), mean)
```

```
##     Site1     Site2
## 0.9098799 1.0438174
```

Parallelize it!

Running `lapply` in parallel

You can gain speed for vector-based operations by using the analogs of the `apply()`-family from the `parallel` package. (There are other packages as well, like `multidplyr` for the TidyVerse.) We'll demonstrate with `mclapply` ("multicore lapply") , but note that there are others as well (e.g., `parSapply`). (*Windows users:* `mclapply` may not work for you, so skip it and move on to the next example!)

```
# Generate a large amount of demonstration data
n <- 1E8
data_list <- list("A" = rnorm(n),
                  "B" = rnorm(n),
                  "C" = rnorm(n),
                  "D" = rnorm(n))

# Single core
system.time(
  means <- lapply(data_list, mean)
)
```

```
##    user  system elapsed
##   1.910   1.899   3.877
```

Compare that runtime to using `mclapply` on twice the number of cores:

```
library(parallel)
detectCores()
```

```
## [1] 4
```

```
cores <- 2 # use as many as you have, if you'd like,
           # but note that you'll leave less resources
           # for the rest of your computer!

# mclapply may not work on a Windows machine!
system.time(
  means <- mclapply(data_list, mean, mc.cores = cores)
)
```

```
##    user  system elapsed
##   0.004   0.029   1.445
```

```
unlist(means)
```

```
##             A             B             C             D
##  6.083002e-05  9.902476e-05  9.695898e-05 -1.118124e-05
```

(Note that running things in parallel will take longer than will non-parallelized functions for trivial problems; it takes a little bit of resources and time to set things up on all cores.)

Running `for`-loops in parallel

When you can't avoid using `for`-loops, but each round of your loop is independent of the others (or you've got nested loops that are independent), you can use the `foreach` package which supports a parallelizable operator `dopar` from the `doParallel` package.

```
library(parallel)
library(foreach)
library(doParallel)
```

```
## Loading required package: iterators
```

```
detectCores()
```

```
## [1] 4
```

```
cores <- 2
cl <- makeCluster(cores) # Create cluster
registerDoParallel(cl) # Activate clusters
system.time({
  means <- foreach(i = 1:length(data_list),
                           .combine = c) %dopar% {
                             # replace c with rbind to create a dataframe
                             mean(data_list[[i]])
                             }
})
```

```
##    user  system elapsed
##  30.979  11.173  84.306
```

```
stopCluster(cl) # Stop cluster to free up resources
means
```

```
## [1]  6.083002e-05  9.902476e-05  9.695898e-05 -1.118124e-05
```

In this case it's slower even than base `lapply`, but that's usually an exception. Nevertheless, it's important to not assume that running things in parallel will always be faster. For example, moving lots of data between cores slows things down significantly.

Profiling

As you can hopefully sense already, there are many ways to write faster code. The more you practice vector-based thinking, the faster, more compact, and – generally speaking – easier to read your code will be come (as long as you don't overdo it). That said, you also don't want to fall into the trap of wasting time trying to speed up code that doesn't take that long to run in the first place! Rather, you want to spend your time optimizing the part of your code that actually does slow things down. This is where profiling comes in.

There are a few packages available (e.g., `lineprof` and `profvis`), but here we'll only use `Rprof()` from base-`R` to identify sections of code (components of a function) that are slowing it down. To demonstrate, let's create a function that has a few functions nested within it.

```
SimCalc_growth_rates <- function(n){
  N0s <- rlnorm(n)
  Ts <- round( rnorm(n, 50, 10), 0)
  data <- mapply(geom_growth_preallocated,
                 N0 = N0s,
                 T = Ts)
  growth_rates <- lapply(data, calc_growth_rates)
  mean_growth_rates <- sapply(growth_rates, mean, na.rm=TRUE)
  return(mean_growth_rates)
}
```

You use `Rprof()` by initiating it above, and ending it below, your code:

```
Rprof(line.profiling=TRUE)
  out <- SimCalc_growth_rates(1000)
Rprof(NULL)

summaryRprof()
```

```
## $by.self
##                 self.time self.pct total.time total.pct
## "rnorm"              0.26    59.09       0.26     59.09
## "<Anonymous>"        0.08    18.18       0.34     77.27
## "mean.default"       0.08    18.18       0.08     18.18
## "FUN"                0.02     4.55       0.10     22.73
##
## $by.total
##                          total.time total.pct self.time self.pct
## "block_exec"                   0.44    100.00      0.00     0.00
## "call_block"                   0.44    100.00      0.00     0.00
## "eval"                         0.44    100.00      0.00     0.00
## "evaluate_call"                0.44    100.00      0.00     0.00
## "evaluate::evaluate"           0.44    100.00      0.00     0.00
## "evaluate"                     0.44    100.00      0.00     0.00
## "handle"                       0.44    100.00      0.00     0.00
## "in_dir"                       0.44    100.00      0.00     0.00
## "knitr::knit"                  0.44    100.00      0.00     0.00
## "process_file"                 0.44    100.00      0.00     0.00
## "process_group.block"          0.44    100.00      0.00     0.00
## "process_group"                0.44    100.00      0.00     0.00
## "rmarkdown::render"            0.44    100.00      0.00     0.00
## "SimCalc_growth_rates"         0.44    100.00      0.00     0.00
## "timing_fn"                    0.44    100.00      0.00     0.00
## "withCallingHandlers"          0.44    100.00      0.00     0.00
```

```
## "withVisible"                0.44    100.00      0.00     0.00
## "<Anonymous>"                0.34     77.27      0.08    18.18
## "mapply"                     0.34     77.27      0.00     0.00
## "rnorm"                      0.26     59.09      0.26    59.09
## "FUN"                        0.10     22.73      0.02     4.55
## "lapply"                     0.10     22.73      0.00     0.00
## "mean.default"               0.08     18.18      0.08    18.18
## "sapply"                     0.08     18.18      0.00     0.00
##
## $sample.interval
## [1] 0.02
##
## $sampling.time
## [1] 0.44
```

```
summaryRprof(lines='show')
```

```
## $by.self
##          self.time self.pct total.time total.pct
## <text>#9      0.34    77.27       0.34     77.27
## <text>#8      0.08    18.18       0.08     18.18
## <text>#2      0.02     4.55       0.02      4.55
##
## $by.total
##          total.time total.pct self.time self.pct
## <text>#9       0.34     77.27      0.34    77.27
## <text>#4       0.34     77.27      0.00     0.00
## <text>#8       0.08     18.18      0.08    18.18
## <text>#2       0.02      4.55      0.02     4.55
## <text>#7       0.02      4.55      0.00     0.00
##
## $by.line
##          self.time self.pct total.time total.pct
## <text>#2      0.02     4.55       0.02      4.55
## <text>#4      0.00     0.00       0.34     77.27
## <text>#7      0.00     0.00       0.02      4.55
## <text>#8      0.08    18.18       0.08     18.18
## <text>#9      0.34    77.27       0.34     77.27
##
## $sample.interval
## [1] 0.02
##
## $sampling.time
## [1] 0.44
```

Note that `Rprof()` is not always able to identify all the components (e.g., when functions aren't named). In these instances it'll indicate said function(s) by calling them "Anonymous".

Faster Computing - Part 2

November 24, 2020

Contents

RStudio Server Pro **2**

High Performance Clusters **3**

`sftp` file transfer 4

`ssh` communication and initial `R` setup 4

`ssh` navigation 5

`ssh` job submission 6

Job completion and ending a job 7

Today we'll be interacting with Cosine's *RStudio Server Pro* and with their *High Performance Cluster* (HPC). In order to access either of these from off-campus, you'll need to be connected to OSU's Virtual Private Network (VPN). OSU uses the *CISCO AnyConnect* client, so if you haven't already installed it, do so now. See download and instructions at https://oregonstate.teamdynamix.com/TDClient/1935/Portal/KB/ArticleDet?ID=76790. You'll need to confirm your access privileges using *Duo* and your ONID credentials.

In order to move files between your computer and the servers, you'll also need an `sftp` client, like https://cyberduck.io.[1]

Finally, in order to pass commands to the HPC, you'll need an `ssh` client. Mac users can use *Terminal*, which is built in to the operating system. Windows users will need to install one (e.g., https://mobaxterm.mobatek.net, which also does `sftp`).

RStudio Server Pro

This one's super simple in that all you have to do is use your browser and ONID credentials to login to

https://rstudio-1.cosine.oregonstate.edu.

If nothing happens, make sure you're logged into the VPN first. Once you're in, you'll be presented with a screen very much like the standard *RStudio* interface with your "*R* session" already initiated.[2] You can start typing commands into the console just as you would on your computer. Of course, you're unlikely to want to do that as you've probably got your code all written and tested on smaller datasets/simulation runs (i.e. you are using the server not for writing or debugging, but just to run code).

To copy files between your computer and the server, `sftp` into the server. In *Cyberduck*:

1. click on "*Open Connection*",
2. select "*SFTP (Secure File Transfer Protocol)*",

[1] File transfer can also be done by `ssh` command-line, but there are no commands to remember with `sftp`.

[2] If you wish to initiate additional sessions (e.g., to run additional code at the same time), click on the blue *R* icon in the top-left corner.

3. type in the Server address:
   `sftp://rstudio-1.cosine.oregonstate.edu`
   (the default *port 22* should be fine),

4. enter your ONID username and password,

5. leave *SSH Private Key* as "*None*", and

6. check the *Add to Keychain* box in order to save your credentials.[3]

You should now be in your `home` folder (e.g., `/home/novakm/`). Take a moment now to bookmark this connection. (In *Cyberduck*, select the *Add Bookmark* option from the drop-down menu.)

In your `home` folder you should see a folder named *R* and, depending on your settings, several "hidden" files and directories.[4] Enter the *R* folder and place whatever files you want inside of it by dragging them in. Be sure to maintain your project's directory substructure so that all relative links between scripts and data work! In fact, you may want (or need) to put much of your whole repository in place (i.e. have your `data`, `code`, and `output` folders and contents all present).

After you've put your scripts in place, jump back to your browser and you should be good to go with a whole lot more computational power at your disposal.

High Performance Clusters

High Performance Clusters (HPCs) consist of many (often hundreds or thousands) of servers that are all networked together. Each server is called a node. You can choose to work on only a single node (which will probably still be faster than your computer), or on several nodes in parallel. The point of using several nodes is that you can use them in parallel (just like we did last class when we used own computer's cores in parallel). The HPC we'll use today is found at `submit.hpc.cosine.oregonstate.edu`.

[3] The option and label for this check-box may be specific to Macs.

[4] The filenames of hidden files and directories all start with a period (e.g., `.bash_profile`). Don't worry if you don't see them as you'll rarely if ever need them. In fact, you typically don't want to delete them, so be careful if you do see them.

`sftp` file transfer

Let's start by using `sftp` to copy files to the HPC.[5] After logging in to OSU's VPN, you can go through the same steps as for the *RStudio Server* except that the server address is:

`sftp://submit.hpc.cosine.oregonstate.edu`.

If the login was successful, you should be in your `home` folder. Again, take a moment to bookmark the connection. This `home` folder will also have a bunch of hidden files (e.g., `.bash`) that you can ignore if they're visible[6], but it may also look empty. Create a folder and copy your scripts into place, again ensuring that your directory structure is correct so that all specified paths will work as in you project's repository.

`ssh` communication and initial `R` setup

Now switch over to your secure shell (`ssh`) client (e.g., *MobaXTerm* for Windows users, or *Terminal* for Mac users[7]). At the $ prompt, type in

`ssh` *`yourONID`*`@submit.hpc.cosine.oregonstate.edu`

whereupon you'll be prompted for your ONID password. You should then see a welcome screen with a new prompt at the bottom of the window:

`[novakm@head ~j]$`

You're likely going to be using *R* a lot, so let's install it into your user resources automatically so it's available each time you login. You can do that by typing the following at the prompt:

`module initadd R`

If you're going to need an *R* package, you'll first need to install it into your home directory. First load the *R* module by typing-in

`module load R`

Then launch *R* by typing-in

`R`

Then type

`install.packages(''`*`package_name`*`'')`

[5] Note that, rather than using `sftp`, you could also use `ssh` and the command `cp ~/Desktop/filename ~/filename` to copy from your desktop to the server.

[6] You can alter *Cyberduck*'s preferences to hide/show the hidden files.

[7] If you want, you can access *Terminal* from within *RStudio*; the tab for it should be in the same window as the console.

The first time you run the `install.packages()` command, you'll be asked if you want to create a personal library. Answer by typing in `y` for yes.
Notice that you're actually in an *R* session. You could, therefore, work away as you might on your computer (but without the benefit of writing scripts).[8]
To get out of *R* and back to the `ssh` prompt, type `q()` for quit.

`ssh` navigation

Now lets find the files we uploaded using the `ssh` view of the cluster. Back at the `ssh`, use the list (`ls`) command:

`[novakm@head ~j]$ ls`

The `ls` command will show you all the sub-directories and files within the directory you're currently in (which at this point in time is your `home` directory). You should see the files (or the folder) you uploaded and, if you installed any *R* packages, a folder named `R_libs`. In order to enter a subdirectory, use the change directory (`cd`) command followed by the name of the directory you'd like to enter:

`[novakm@head ~j]$ cd` *subdirectory*

You can move down into multiple nested dirctories, e.g.,

`[novakm@head ~j]$ cd` *subdirectory/subsubdirectory*

and move out of any number of directories, e.g.,

`[novakm@head ~j]$ cd` *../../otherdirectory*

just as we did when setting relative paths in *R*.

A very useful feature here is that you don't need to type out the whole name of a directory or file. Just type the first letter of its name and then hit your tab key. The name will autofill until it gets to a letter where it can't distinguish between similarly named files. Type the distinguishing letter and tab to continue until you've got the whole name.

Once you're in the directory in which you'd like to be, type `ls` to confirm all the contents are there as needed. If you'd like to take a quick look at the contents of a file, you can use the concatenate (`cat`) command:

`[novakm@head ~j]$ cat` *file.r*

The contents won't be rendered very nicely, but the function is nonetheless useful for confirming the contents of short scripts (such as the `submit.sh` submission script that we'll talk about below).

[8] You can use such command-line interfaces for *Matlab* or *Mathematica* too. In fact, there are many, many other such "modules" available on the HPC. Just type `module avail` to see the whole list.

`ssh` job submission

When performing analyses on an HPC, you (typically) don't do so be entering into the module (e.g., into *R*, the way we did above when installing *R* packages). Rather, you use a submission script to submit your code (your "job") to the cluster. The primary reason for doing that is that the cluster has extensive automated job managment tools which optimize the use of nodes among users and nodes. Therefore, when you submit a job, the cluster (typically) determines which of its nodes it will send the job to. It's similarly so when your job contains code that performs parallelized computations.

The submission script for an *R* job on our HPC will look like this[9]:

```
#!/bin/sh

# Job name (replace R_simple but leave -N)
#$ -N R_simple

#$ -S /bin/sh

# Set working directory on all host to
# directory where the job was started
#$ -cwd

# Send output to job.log (STDOUT + STDERR)
#$ -o job.log
#$ -j y

# Email information (to receive email at process end)
#$ -m e
#$ -M username@oregonstate.edu

#Change which version of R you want to load
module load R/3.3.1

# Command to run (replace test.r but leave Rscript)
Rscript test.r
```

[9]Examples for a *Mathematica* submission and an *R* array submission (distributed over multiple nodes) is provided in the `HPC_examples` folder of today's class folder.

Copy the above into a text file, edit it as needed (remembering especially to give the job an informative name at the start and change the script it runs at the end), and save the file with an `.sh` extension (e.g., `submit.sh`).[10] Now upload this file into the same directory as the `test.r` script it calls using your `sftp` client (e.g., *Cyberduck*). Ensure it's in place using your `ssh` client.

Let's assume that our `test.r` analysis script and our `submit.sh` submission script are located in a directory named `mytest`. Use `cd` to change into the directory and `ls` to ensure both your script and your submission script are present.

To submit your job, use the `qsub` command:

```
[novakm@head mytest]$ qsub submit.sh
```

There are a few commands with which you can view and confirm that your job is running. Each offers different and different amounts of information.

`[novakm@head mytest]$ qstat -u` *username*

You'll see the "state" of your job in the far right column[11]. The most common states for a process to be in are "q" (qeued), "r" (running) and "s" (suspendend), and "e" (error)[12]. To see how much of the cluster's resources your jobs are using, type

`[novakm@head mytest]$ top -u` *username*

For more abbreviated views, use `ps` (for all of your running processes) or `jobs` (for all of your current running jobs).

Job completion and ending a job

If you typed-in your correct email address in the submission script, you'll get an email sent to you when it completes successfully or ends in an error. A log of your job will also be saved to the output file you specified in your submission script (which can be useful for debugging). The log will include output that your script printed to screen (i.e. what would have appeared in your *R* console had you run the script on your computer). Use your `sftp` client to grab all the output your script produced.

[10] You can give it any name you want, just give it the `.sh` extension.

[11] To have a "live" view rather than a single snapshot, use `watch qstat`. Use command-key period to escape.

[12] To get more information, use `qstat -j`.

Should you decide that something isn't working right for an active job (e.g., a job is taking far longer than expected, eating up too much of the cluster's resources, or you realize you do in fact have a bug in it), you can end it prematurely using the `qdel` command:

1. get its *process ID* (PID) from the left hand column of the `top -u` view;
2. return to the command prompt (using your command key, e.g., Apple key or Windows key, followed by the period key);
3. type `qdel` followed by the process ID.

If `qdel` doesn't work, replace `qdel` with the nuclear option: `kill`.

Finally, to close your `ssh` session, use the `exit` command.

Command	**Action**
Command-key period	Return to command prompt (end current view)
`qstat -u` *username*	View status of submitted jobs
`top -u` *username*	View resource usage by job
`qdel` *process id*	Stop and delete a submitted job
`kill` *process id*	Stop and delete a submitted job (nuclear option)
`exit`	Close your `ssh` connection to the HPC