Analytical Workflows

All course notes strung together

September 19, 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

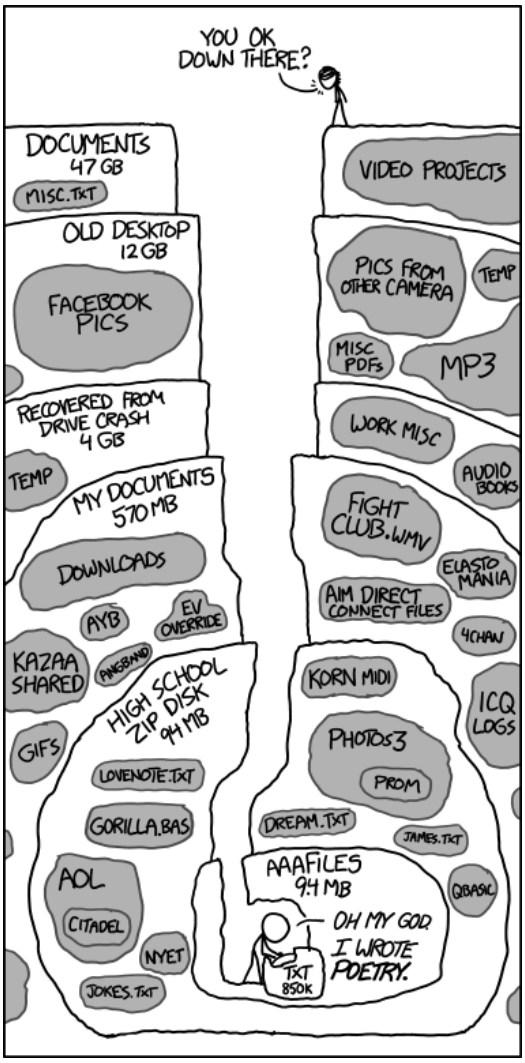


Figure 1: Old files (source: http://xkcd.com/1360/)

Getting started with `Git` & `GitHub`

Contents

What is version control?

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 now 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 (though not totally so), especially 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 also 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`) rather than command-line.

Github

`Git` stores a complete copy of the project on your local machine, including all its history and versions. That means that 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

[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.

is perhaps a little easier to work with. The two don't differ all that much 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 log in to your `Github` account, click on "New Repository", and follow the instructions. These should include options for private versus public (pick the latter for this class), initializing with a `README` file (which you *should* do), and adding a `.gitignore` file (which you *should* also do).

The `README` file is the first file that anyone will see when they inspect your repository (assuming it's public). 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 files later in the course, so for now just write some minimally informative text in it without any formatting (e.g., your project title).

The `.gitignore` file contains a list of all the files that you want `Git` to ignore (not monitor for changes). Selecting `R` from the dropdown list will auto-fill a bunch of it for you. Later in the course, we'll also add the extensions for all the temporary files that LaTeXproduces 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. As I said earlier, there's a lot of bells-and-whistles at your fingertips here (the most useful of which for collaborative projects is the `Issues` feature). 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 (from your projectfolder) and go from there.

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;

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

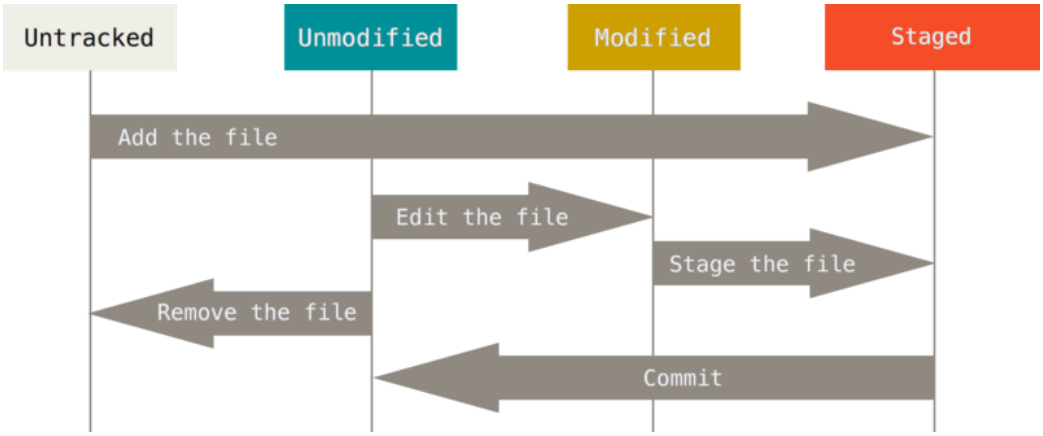


Figure 1: The `Git` life-cycle.

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.[3]

`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 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

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

updated your local respository. 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 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.[4]

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

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

[4]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>*

Best practices

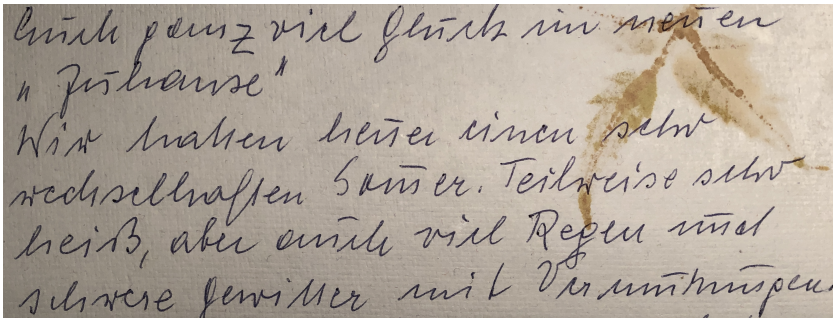
September 18, 2020

Contents

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')`

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.[4] 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')`

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

```r
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 DIY 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 DIY 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.

```
#  sd -- the standard deviation of gaussian noise (
   default=0)
# Returns:
#  Vector of length equal to the input vector

add_noise <- function(x, noise_model='gaussian', sd=0){
   noise_model <- match.arg(noise_model)
   if(noise_model=='gaussian'){
      out <- x + rnorm(length(x), sd)
   }else{
      out <- x
      warning(''Original time series returned'')
   }
   return(out)
}

# Test:
# source('simulateLV.R')
# new <- add.noise(out, sd=1)
# par(mfrow=c(1,2))
#  plot(out)
#  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 (visiually) 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 mis-specified. 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).

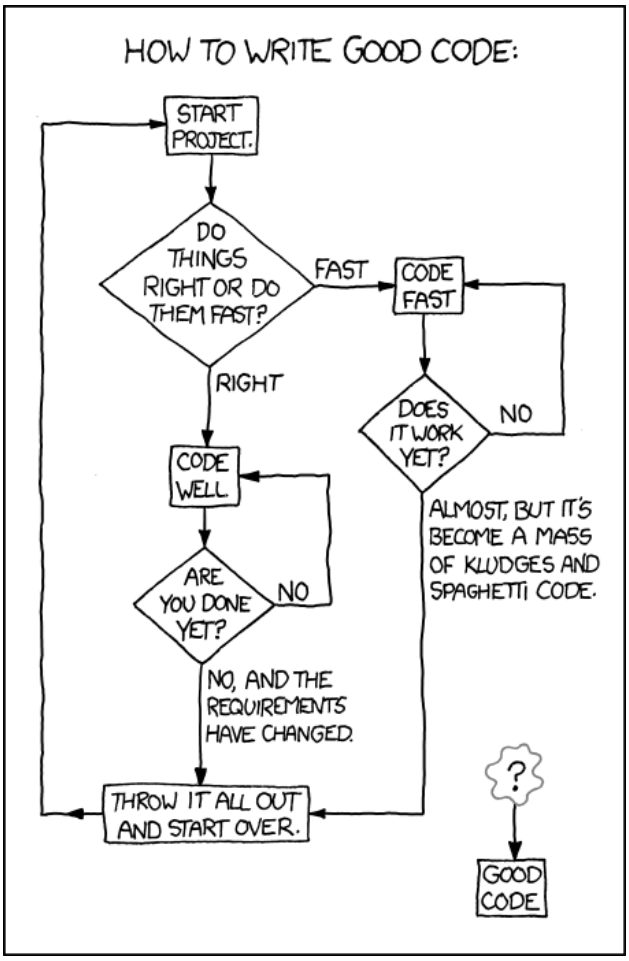


Figure 1: Good code? (source: https://xkcd.com/844/)

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 includes 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 Google's Style Guide for R (**required reading**) before returning to continue reading here.

https://google.github.io/styleguide/Rguide.xml

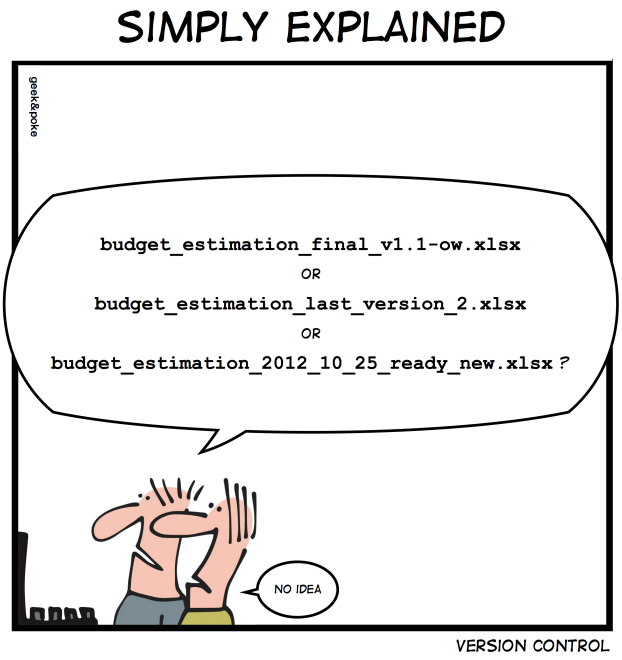


Figure 2: Your data files *will not* look like this; use `Git`!

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.[7] 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.[8]

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.

[7] 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.

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

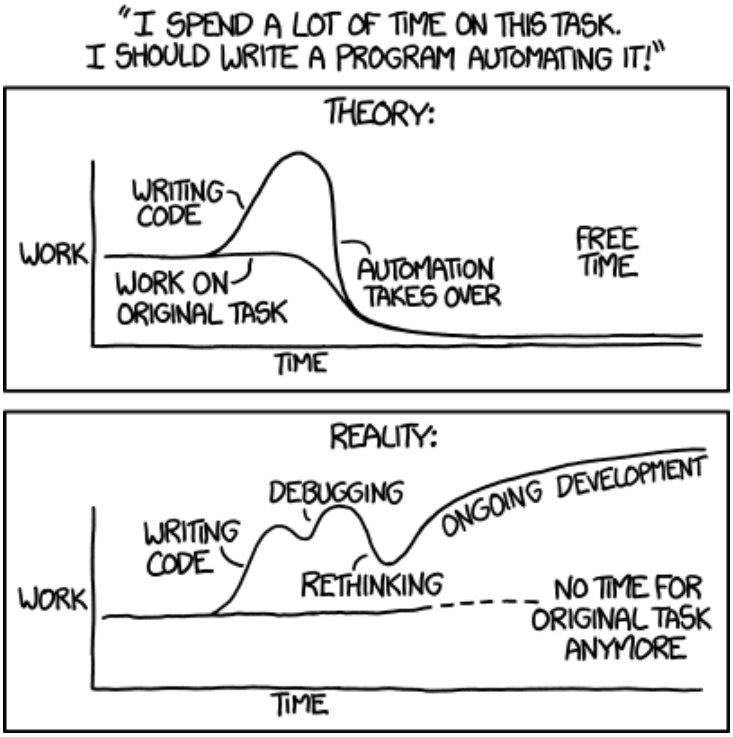


Figure 4: Admittedly, sometimes reality is indeed more like this. (source: https://xkcd.com/1319/)

More with `Git` and `GitHub`

Contents

Commit early, commit often. Pull, edit, commit, push.

Almost all of what we'll talk about below can be done using a `Git` GUI and on `GitHub`, but for generality I'll most often describe actions using the command-line process. That should still allow you to get the gist of what to do.

What not to commit

What is a "flat" file?

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)

Merging

Resolving merge conflicts:
https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/resolving-a-merge-conflict-using-the-command-line

Something to consider when merging is your merging pattern. That is, the most obvious choice is to merge your new feature branch (the one in which you've been making changes) 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, testing that everthing works, then merging back into the master. This pattern reduces the risk of introducing new bugs into 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 the latter approach is that it takes more time and results in double the number of merge commits. An alternative that solves this problem is rebasing, but that comes with other potential challenges.

Recovering from Git errors

Reverting to a previous commit

Say you just committed somethign but want to look back at your previous commit. Just use `git checkout` followed by the commit #. Alternatively, `git checkout HEAD^`, where the carrot refers to the commit prior to the current commit.

If your last commit was made in error and you want to undo it, you have a couple options:

To undo the last commit completely, use `git reset --hard HEAD^`. This rolls back your repository to the previous commit; changes not reflected in the commit-before-last will be lost forever.

To undo the commit but leave the files in that state but *unstaged*, use `git reset HEAD^`. This rolls back your repository to the previous commit, with any changes not reflected in the commit-before-last remaining. The last

commit will be undone and nothing will be staged.

To undo the last commit but leave the files in that state and *staged*, use `git reset --soft HEAD^`. This rolls back your repository to the previous commit, with any changes not reflected in the commit-before-last remaining as staged changes.

Finally, if you just want to make a small change to the last commit you made (e.g., you forgot to include a file, had a minor typo somewhere, or want to change the commit message), you can `amend` the last commit without creating a new commit using `git commit --amend -m "New commit message"`. Be careful though as this is changing history! Also, once you push a commit to GitHub, you cannot amend it. Doing so will create an error the next time you try and push to GitHub.

Recovering from mishaps

The last push is your worst case recovery scenario (see Fig. 2): when all else fails, you can always delete your entire local repository and re-clone from GitHub to pick up at the last pushed commit.

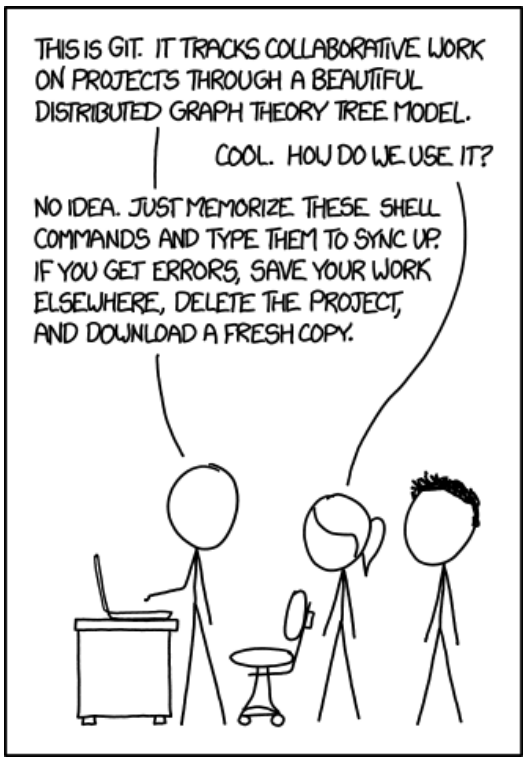


Figure 2: Solving errors in Git (source: https://xkcd.com/1597/)

This will be my article title.

An Intro to LaTeX

September 14, 2020

Contents

Figure 1: This is the LaTeX logo.

Introduction

You can do all the following without a LaTeX gui using https://www.overleaf.com. For writing one-off equations use LaTeXiT https://www.chachatelier.fr/latexit/. To find a symbol: http://detexify.kirelabs.org/classify.html.

My un-numbered subsection

The fundamental principle of calculus entails

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

Calculus rocks because it can do all of the following:

- first item

One could also organize all the cools things it can do in a table

Reason	Explanation
1	blah blah blah

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

1 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).

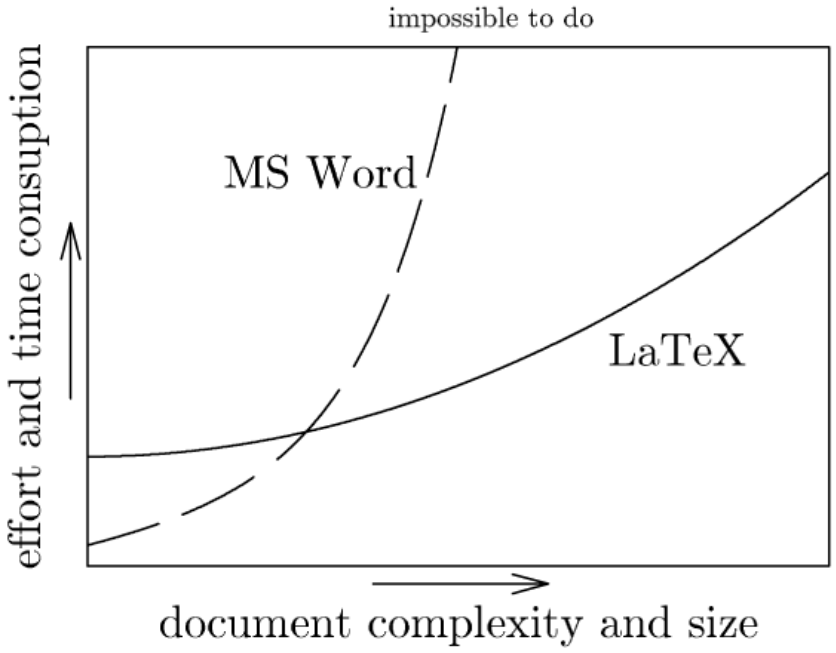


Figure 2: Why use LaTeX (source: http://www.pinteric.com/miktex.html)

2 Some other stuff

Here are a few sketches for equations that might come in handy.

Differential equations

The Lotka-Voltera equations are given by

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

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

These can be written "inline" using (for example): $\dot{x} = \alpha x - \beta xy$ etc.

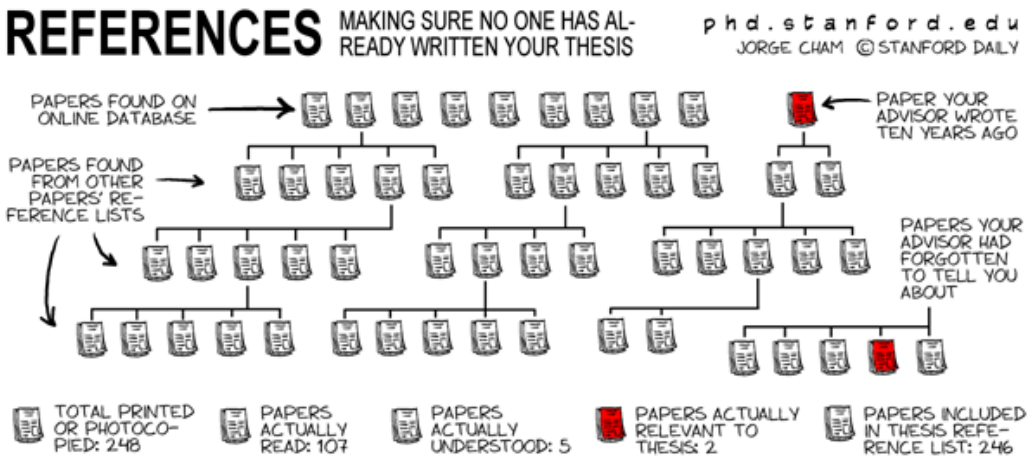


Figure 3: Reading is fundamental (source: http://phdcomics.com/comics/archive.php?comicid=286)

Derivations

It is nice 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.

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

3 References

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 but 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 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 for ourselves and our families, and not become food for another creature. Think of this like the GPU (graphics processing card) of your brain-computer. It is soooo much more powerful than your abstract processing abilities (could think of this like the CPU of the computer). We can do better science by working on our GPUs...

Example of a basic visualization workflow

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

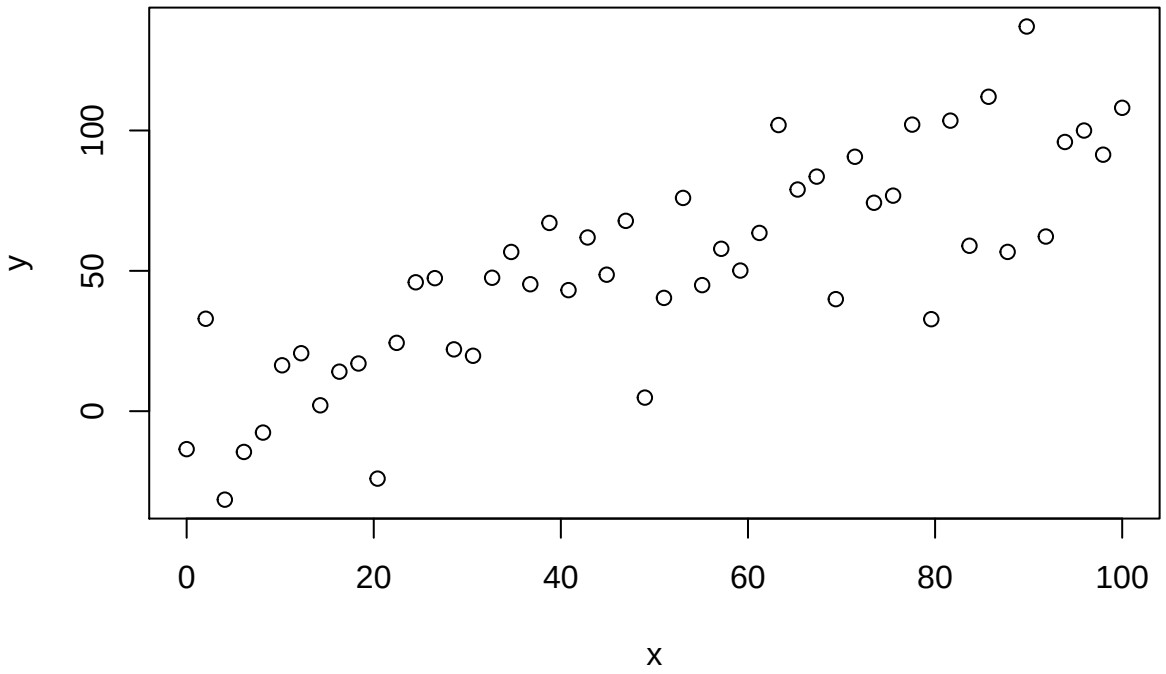
Setup

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
```

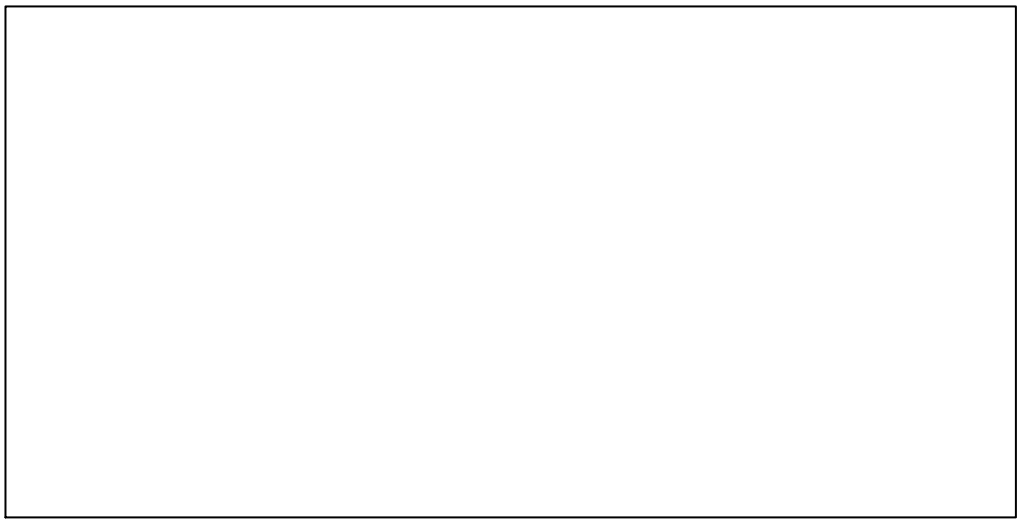
At this point it is worth spending some time in the help files for `plot()` and `par()` if you have not already. This is just to get a sense of the potential to control things, and how the basic control process works. Most people need to continually refer to these help files as they work on specific 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 ned to be big 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 an 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 readers 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, hardcoding 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, 120))
```

Tune for impact

Now that we have something that meets minumum formatting 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: "ie 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)                    #quick way to add 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 predicitons, then plot that. First generating the predictions

```
npred <- 10
xpred <- seq(min(x), max(x), length.out = npred)                          #points at which we want to pre
ypred <- predict(fit, newdata = data.frame(x = xpred), se.fit = TRUE)    #predicting corresponding y val
```

Now plotting:

```
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)
```

It is often easy to reproduce tidyverse functionality in base R. Let's make our plot look like ggplot

```
# 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 predictable way." What about secondary messages? Suppose we are interested in what caused the population to grow especially quickly or slowly in some years. Let's use symbol color and shape to highlight points that are outside the confidence envelope.

To start with, it will be helpful to have the model prediction and standard error for each observation point, rather than at 10 evenly spaced points as it was before

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

Now let's identify points that are boom and bust years, meaning those above and below the confidence envelope for our fitted model

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

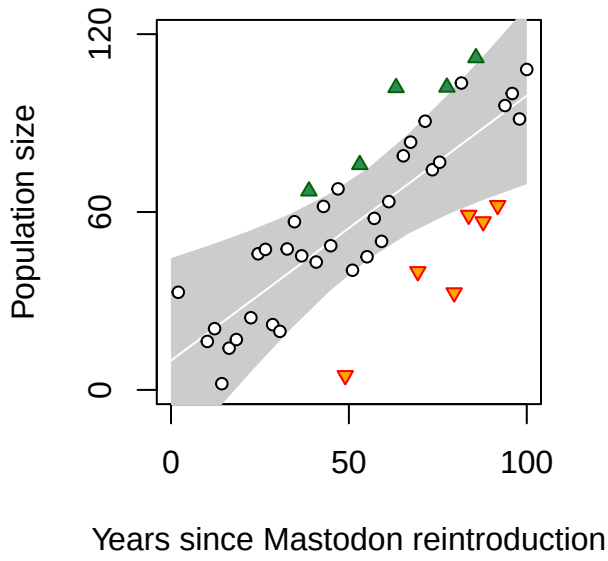
I like the following procedure when working with symbology. First specify size, shape and color as vectors, mapping from data. Then we call plot function. I think this is better than doing the mapping inside the plot function call, because that is hard to read and debug.

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


# Change them to highligh features in 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 ever 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.

```
# 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 predictable way."
2. "We are going to focus on the years where growth was particularly fast or slow relative to those predictions"

Future graphs could use the same color scheme and symbology to denote analyses pertaining to the boom and bust years.

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.