PHO Data Management Guidelines

Callum Arnold

January 21, 2019

Reproducible Work

What is reproducible work?

- Contains relevant code, including which packages were used
- Contains enough text, either via `markdown` or comments, to be able to understand what the purpose of each code is
  - Ideally integrates code and results, along with text, into a single document

Why is it important?

Simply put, mistakes happen. If your project if structured properly, you will have a code document that contains all of the relevant information, and it is easy to recreate the outcomes. That way, if you move computers, delete a document by accident, or hand over the project to another person, everything is neatly contained and can be reproduced without hassle.

How do I make reproducible work?

There are many different ways to make reproducible work. The information listed here should give you the foundations upon which you can build your own systems. However, the principles are the same, and largely revolve around project structures and a version control system, such as `git`.

Structuring a project

This is the structure that I find works for me. You may want to find a variation on it that works for you, but the basic premise of keeping repositories self-contained should remain.

```
proj/
  data/
  docs/
  figs/
  funs/
  out/
  cleaning.R
  analysis.R
```

As you can see, the project repository contains separate directories that you can use to store different file types. Importantly, the analysis and cleaning files are stored in the project root, allowing easy use of relative paths over explicit paths e.g. `read_csv(here('data', 'data_file.csv'))` rather than `read_csv('C:/Users/owner/Documents/Repos/my_project/data/data_file.csv')`. The reason why we prefer relative paths is that they allow projects to be used by multiple people without the need to re-write code. If you change computer, or the project is opened by another person, the code will break as they will not have the same directory structure as the computer that the code was created on.

> **Note:** the example above used an `R` package called `here_here`, calling the function `here()`. Similar solutions may exist for other languages, and you should try and find them for the language of your choice.

`data/`

An important idea is that you should treat your data as read-only. You and your team have likely worked hard to collect the data and it's easy to make a changes along the way that you either forget about, or need to reverse. As most projects span a long time between the data collection and analysis stages, if that happens to you it will take a lot of work to figure out exactly which changes you are interested in reversing etc. To save yourself this hassle, and help make your work reproducible, once the data is collected it should not be edited; all the work should happen in your code, allowing it to be easily checked.

Other subdirectories

- `docs/`: this contains the output documents. For example, if you are using `R Markdown` to create a pdf via `LaTeX`, you could place them here.
- `figs/`: this contains the functions you write and might want to reference. The idea is to create functions so that can give code a meaningful name. It also helps if you need to repeat a code chunk multiple times, especially if you need to edit it at some point, as you can just call the function rather than typing it out each time.

- `out/`: this contains files that are produced from the original data e.g. cleaned data files. You can then call them in your analysis scripts.
- `figs/`: this contains figures that may be generated from your scripts.

Importantly, if you follow the principle that your `data/` files are read-only, all of the files in these directories (with the exception of `funs/`) *should* be reproducible and could be deleted at any time without concern of generating them again. In order to revert to previous figures and output versions, you will need to be able to track changes in your code. This is where a *version control system* like `git`, which we will discuss in the next section.

File names

How you name files and directories may not seem like an important point, but it can cause quite a headache if you try and use code to automate processes, and at best, it just slows things down. To quote Aaron Quinlan, a bioinformatician, "a space in a filename is a space in one's soul".

Instead try and use something like this.

- KISS (*Keep It Simple Stupid*): use simple and consistent file names
  - It needs to be machine readable
  - It needs to be human readable
  - It needs to order well in a directory
- No special characters and **no spaces**!
- Use YYYY-MM-DD date format
- Use `-` to delimit words and `_` to delimit sections
  - i.e. `2019-01-19_my-data.csv`
- Left-pad numbers
  - i.e. `01_my-data.csv` vs `1_my-data.csv`
  - If you don't, file orders get messed up when you get to double-digits

Key Points

- Use a version control system such as `git` to track changes in your code.
- Data isn't touched one collected:
  - Do all *data munging* within your program i.e. no editing the excel spreadsheets!!!
- Never set explicit file paths (e.g. `setwd()`) if you can avoid it
  - Try and use a package that allows you to set relative paths e.g. `here_here` in `R`. This allows the project to be passed to someone else in its entirety and the code won't break because they don't have the same folder names and set up as you (also if you work on multiple computers/OS)
- Format your filenames properly

Git

Think of it as tracked changes for your code. When working on a project, even if you're the only one coding, it's important to be able to go back to previous versions if you make a mistake and can't remember all the steps you went through since your last stable version.

Set up

There are many ways to get `git` running on your computer. Depending on your OS and the version you have, `git` may come pre-installed on your computer. However, it is a good idea to update it to the latest version, so I'd recommend you follow the steps below anyway.

Github

This section will need to be updated after figuring out the PHO system

It is not essential, but one of the best things about `git` is that online repositories create an easier way to sync your work between computers and colleagues, avoiding much of the mess caused when this work happens simultaneously/(insert file sharing system of choice) isn't syncing properly.

With Github now offering unlimited free private repositories, I would recommend that you set up an account with Github. It's very easy to register. Just click the link above and select the package you'd like.

If you have an academic email address, consider making this your primary email address on the account, as it gives you a **PRO** account unlimited collaborators on private repositories, unlike the standard account that limits it to 3 collaborators.

Be sure to choose a user name that is easy to remember, and easy to find. I would suggest just using your name.

Now you have a Github account set up, this is your *remote*. If you work on a project with collaborators, this can be shared with them. That way, collaborators can work on their own versions of the code on their *local* machine (computer), and when it's ready for other people to use/help write, you can `push` it to the *remote* where other people can access it. Don't worry if you don't know what `push` is - we'll cover that once we've got `git` installed.

Windows

1. Install Git for Windows

- This gives you `git bash`, which is a much nicer way of interfacing with `git` than the command line.
- **Note:** when asked about "*Adjusting your PATH environment*", be sure to select "*Git from the command line and also from 3rd-party software*". The other default options should be fine. For more details about the installation settings, please click here

2. Open up `Git Bash` and enter `where git`. Open up the command line and enter `where git`. Depending on whether you have administrator privileges, the outputs should look something like this, respectively
    1. `which git` : `/mingw64/bin/git`
    2. `where git` : `C:\Users\owner\AppData\Local\Programs\Git\bin\git.exe` (User privileges)
        1. `where git` : `C:\Program Files\Git\bin\git.exe` (administrator privileges)
    - If you see `cmd` instead of `bin`, then you need to edit the `PATH` in your environment variables. You can do this by typing `environment variables` into the Start box and scrolling to the `PATH` section of User/System variables (depending on whether you have administrator privileges), and changing `cmd` to `bin` in the `git.exe` path.

Mac

There are more (workable) ways to install `git` on OSX than on Windows, but I think this is the best option as it gives you a great package manager for the future.

1. Open the terminal and enter `/usr/bin/ruby -e "$(curl -fsSL https:/raw.githubusercontent.com/Homebrew/install/master/install)"`
2. Enter `brew install git` into the terminal

Final set up steps

Now that you have `git` running, you need to tell it who you are. This allows multiple people to make changes to code, and the correct names will be attached to the changes.

Open up the `Git Bash` and enter

```
git config --global user.name 'Firstname Lastname'
git config --global user.email 'my_email@gmail.com'
```

Typing in `git config --global --list` is a way to check that your details have been saved correctly.

Note: it is essential that you enter the same **email** as your Github account information. This way you can connect the two. If you would prefer to use a different user name than your Github user name you can. This would help show you which computer you completed the work on, but it is not important to most people.

Installation problems

If you followed the instructions above, `git` should be ready to go. However, sometimes you still end up with errors. This is far more likely with Windows that Mac, but if you find that the next steps don't work for you, see if the other installation options here work for you, or the trouble shooting tips here (and here if you're on Windows), which are useful when trying to connect `git` with `RStudio`

Git client

Now you have `git` installed, there are a number of ways to use it. The easiest way is to use a client, which allows you to use buttons instead of typing code. They also provide a visual for more complicated ideas, such as branching, greatly simplifying the process. I prefer to use the SourceTree client, but you can use others.

Creating a repository

This section will need to be updated after figuring out the PHO system

If everything has gone well until now, you're ready to create a project repository. This is where all your code, all your data, all your output files, everything, should live. Whilst you can create a repository directly on your computer, I would advise against this as it causes additional headaches when you want to connect it with Github. Instead, create the remote repository first on Github.

- Go to www.github.com and click the `+` and *"New repository"*.
- Choose a project name
- Decide whether you want it to be a public or private project (choose private if working on PHO data and research, as you can always convert it to public later)
- Initialize with a README file

Connecting to Github

This section will need to be updated after figuring out the PHO system

If you are using SourceTree, there are two ways to connect your computer to your Github repositories. I would recommend the first option, as it makes *cloning* repositories (making a copy on your local computer) easier in the future, as you don't have to go to Github each time to find the *HTTPS/SSH* address.

Method 1

1. Go to *Tools -> Options -> Authentification* in SourceTree and add your Github account details
2. Leave the preferred protocol at *"HTTPS"* for the moment, unless you know what you're doing with SSH keys
3. Now open a new tab, click on *Remote*, and you should see your repositories listed
4. Clone the repositories that you'd like to work on

   Note: Now SourceTree and Github are connected, you shouldn't have to do the first few steps - just go to step 3.

Method 2

1. Go to your repository on Github and click on the green *"Clone or download"* button
2. Copy the *HTTPS* address (your settings should say *"Clone with HTTPS"* above it, otherwise click *"Use HTTPS"*)
3. Go to SourceTree, open a new tab, and click on *"Clone"*
4. Paste the *HTTPS* address into the *"Source Path / URL:"* box
5. Click on the *"Clone"* button at the bottom

Method 3 (*not recommended*)

If you would like to do it the hard way and use the `Git Bash`, read the instructions here

Note: if you would like to use SSH keys, read the instructions here

Basic commands

There are many commands that you could learn in `git`, but these are the basics, and will be sufficient for pretty much everything you'll need to do at the moment.

- `commit`: this standings for *committing* a change to your file in `git`. Think of it as saving a document, but instead of saving the whole document as-is, `git` saves just the changes since the last version. This makes it very efficient, especially when it comes to backing up your work. **Key points:**
  - `commit` often. By making and saving small changes, your code versions becomes more readable in case you need to go back and find out exactly what and where it went wrong.
  - Always write helpful messages - keep them succinct, but make sure they describe what the change you made was.
- `pull`: this commands copies the version of the code from your remote to your local machine. Use this when you want to get the most up-to-date

version of your code to work on (assuming your local version isn't the most up-to-date)
- `push`: the opposite of `pull`. If your local version is the most up-to-date version, `push` your version to the remote.

Branching

Branching is a key part of the `git` work-flow. It allows you to make changes to your code, without worrying about breaking previously 'good' code. But what is it?

Simply put, when you create another branch you are creating a copy of your code at that point in time. This is useful because it allows you to make changes to your copy, and leave your original code intact! So there's no concern about breaking your working code while you test out some ideas.

> "But isn't that why we use git?"

Kind of. But git is only so powerful. If you have working code, you don't want to put it out of action whilst you test ideas out, especially if other people need to use your code and can't wait for you to figure out your future problems. So creating another branch allows you to get around this issue. For most projects, you can get away with just two branches, a `master` and a `develop`, which are explained below. If your project is complex, and requires multiple people to work on the code at the same time, it would be worth you looking at implementing this model.

The image to the right is copied from the model listed above. It is useful in illustrating the `master` and `develop` approach to branching.

Creating a branch

As with all things in `git`, you can do this multiple ways. I prefer to use the SourceTree client, as I find it far more intuitive when you can see the changes, but you can use the command line or git bash. If you want to explore the command line code, I would recommend visiting this website, which allows you to interact with the code through illustrations.

Notebooks

- Many examples of different kinds of notebooks
- Jupyter notebooks are useful as you can see the results immediately integrated within the document

    - Have a system to allow other statistical and programming languages to run via kernels
        - Other systems don't allow this level of integration, so will not be explored in much detail
- Jupyter labs is the updated version of notebooks that acts as a full IDE (integrated development environment), allowing you to

Jupyter notebooks

This section will give you a brief overview of what a Jupyter notebook is and how to use them, but if you would like a more detailed understanding, please read the official documentation.

Jupyter notebooks are run on `python`, though additional things can be downloaded to allow you to use your programming language of choice. For an example of what you can do with Jupyter notebooks, click here, and here for a collection of neat and applied notebooks.

Mac's come shipped with a version of `python`, but it is most likely outdated, and it doesn't contain everything we want. In order to get running, I strongly recommend downloading the `anaconda` distribution over other distributions, or even just directly from `python's` website. The instructions below will be enough to get you up and running with Jupyter notebooks in your language of choice.

- Download the full anaconda distribution i.e. not miniconda
    - Be sure to choose `Python 3.x`, not `Python 2.x`, as it's the newer version and is forwards-compatible.
    - Be sure to only install for one user, not the whole system
    - Be sure to select `Add Anaconda to my PATH environment variable` under Advanced Options
    - Be sure to install `Anaconda` to the `H:\` drive on your computer, as this is where your data lives. To do this you will need to manually edit the installation path within the `anaconda` installer wizard, otherwise it will end up in the `C:\` drive
        - This is OK if you are able to store data on this drive, and therefore can create your repositories within the `C:\` drive.
        - Worst case scenario you can use the command `cd "H:/..."` at the top of the notebook to specify the relevant path to your data, but this is bad practice for the reasons mentioned above.
- Use `kernels` to connect your programming language of choice with python and the notebook
    - To see how to get a particular language to work in Jupyter Notebooks, please click on the appropriate language:
        - Stata
        - SAS
        - R

      * Other kernels
- Visualization for data exploration:
  - plotly (https://towardsdatascience.com/the-next-level-of-data-visualization-in-python-dd6e99039d5e)

The following section is not essential.

Because I do not like the `In[] Out[]` text showing in documents, along with centering plots/figures, I have customized the Jupyter notebook settings. If you would like to do the same, please refer to the section below. It is not necessary, but I feel that it gives cleaner documents (including pdf documents via LaTeX).

If you would like to customize the look of the notebook, jupyterthemes is a great package that can be installed. I have also edited the `custom.css` file (`C:\Users\owner\.jupyter\custom\`), adding `display: None;` under the section

```
div.prompt,
 .prompt {
```

so that it now reads

```
div.prompt,
 .prompt {
 font-family: monospace, monospace;
 font-size: 9pt !important;
 font-weight: normal;
 display: None;
 .
 .
 .
}
```

This removes the `In[] Out[]` text. To center the output of tables/figures, add

```
.output_png {
    display: table-cell;
    text-align: center;
    vertical-align: middle;
}
```

to the `custom.css` file, right after the `.prompt {..}` section.

To enable soft wrapping in the notebook, you need to edit the `notebook.json` file (`C:\Users\callum arnold\.jupyter\nbconfig\`). If it does not exist, you need to create it. Once open, add

```
{
  "MarkdownCell": {
    "cm_config": {
      "lineWrapping": true
    }
  },
  "CodeCell": {
    "cm_config": {
      "lineWrapping": true
    }
  }
}
```

before restarting Jupyter.

Creating a notebook

> **Note:** You can substitute the phrase *"Jupyter notebooks"* with *"Jupyter Labs"* if you would prefer to have a full IDE allowing you more control over the system.

You can either open up the anaconda navigator and then Jupyter notebooks, or open Jupyter notebooks directly. Once open, navigate to the directory you would like to create the notebook in (*If you are using a version control system like git, then you should be within the project's repository*)

Select the **New** button in the top right corner, and then select the language you would like to program in (*this assumes that you have downloaded an appropriate* `kernel` *if you would like to use a language other than* `python`)

Installing the Stata Kernel

The instructions for installing the `stata_kernel` are based from the original documentation here. It should work with `Stata 12` (we have tested it). If these instructions do not work for you, it may be that there has been an update to the `kernel`, at which point, please refer to the original documentation linked above.

Open a command prompt (Windows) / terminal (linux/mac) and type/copy-paste the following commands, pressing enter after each line

- `pip install stata_kernel`
- `python -m stata_kernel.install`

Windows-specific steps

In order to let `stata_kernel` talk to Stata, you need to link the Stata Automation library:

1. In the installation directory (most likely `C:\Program Files (x86)\Stata12` or similar), right-click on the Stata executable, for example, `StataSE.exe`

(this will just show as `StataSE`, but is listed as an application). Choose `Create Shortcut`. Placing it on the Desktop is fine.
2. Right-click on the newly created `Shortcut to StataSE.exe`, choose `Properties`, and append`/Register` to the end of the Target field. So if the target is currently `"C:\Program Files\Stata12\StataSE.exe"`, change it to `"C:\Program Files\Stata12\StataSE.exe" /Register` (note the space before `/`). Click OK.
3. Right-click on the updated `Shortcut to StataSE.exe`; choose Run as administrator.
4. Enter your CIHS details

Installing the SAS kernel

This has not yet been tested here at PHO. The instructions for installing the `sas_kernel` are based from the original documentation here

Open a command prompt (Windows) / terminal (linux/mac) and type/copy-paste the following commands, pressing enter after each line. First we need to install a dependecy called `saspy` that helps the kernel connect `SAS` to `python`

- `pip install saspy`
- `pip install sas_kernel`

You should now see something like this.

```
Available kernels:
python3    /home/sas/anaconda3/lib/python3.5/site-packages/ipykernel/resources
sas        /home/sas/.local/share/jupyter/kernels/sas
```

Now verify that the SAS Executable is correct

- find the `sascfg.py` file – it is currently located in the install location (see above) `[install location]/site-packages/saspy/sascfg.py`. To query `pip` for the location of the file, type `pip show saspy`. Failing that, this command will search the OS for the file location: `find / -name sascfg.py`
- edit the file with the correct path the SAS executable and include any options you wish it include in the SAS invocation. See examples in this file

Connecting R with Jupyter

If you are hoping to make nice documents and reproducible work using `R`, I would highly recommend that you use the `R Markdown` or `R Notebook` through `RStudio` application instead. However, if you would prefer Jupyter, then please read on.

It is possible to download an `R kernel`, much like for `Stata` and `SAS`, but it can be a bit fickle, so a different approach is described below. It is important to note that with this method you are installing a fresh version of `R`, so you will not have access to the packages you have previously installed - you will need to reinstall them in this `R` environment, which could be done within a Jupyter notebook.

Open a command prompt (Windows) / terminal (Linux/Mac) and type/copy-paste the following commands, pressing enter after each line

- `conda install r-essentials r-igraph`
- `Rscript -e 'install.packages("languageserver")'`

If you would rather install an `R kernel` than a fresh install of `R` within the `anaconda` distribution, you can follow the instructions here. The advantage of this is that it allows the notebook to access previously installed packages as they are not running off a fresh version of `R`.

Connecting other kernels

To see a full list of `kernels` available for Jupyter, along with the appropriate documentation and installation instructions, follow this link.

Git and Jupyter

Unfortunately, `git` and Jupyter don't always play nicely, so we have to do a few things to try and get around the issues. Due to the way the notebooks create and store the outputs from the code, `diffs` become unreadable. There are a few ways to get around this. The first option is the simplest, but the others provide a little more control over what you see in the `diffs`.

****Test out the other options!****

1. Clear all outputs before you save and commit the files. That way, `git` only tracks changes to the input
2. nbdime
3. ReviewNB
4. Nextjournal is a promising take on notebooks that simplifies the process of making reproducible research. Currently it is only in beta, and for private research, but if it has a free version when it becomes established it would be a good option allowing a 'Google Docs' style of version control

Output documents

We've covered a lot of information up until now about setting up your projects and your code, but a big part of reproducible research is the creation of nice-looking documents. The reason we've gone to such effort to install Jupyter notebooks and connect them with our language of choice is that not only do they allow for excellent data exploration, but they also make documents that look professional.

In scientific articles, whilst it's not essential, LaTeX is a nice touch, and Jupyter can give you a LaTeX formatted pdf documents. To do this, you will first need to install LaTeX. If you are on Windows, I would recommend the MiKTeX distribution, and if you use a Mac, then I would recommend the MacTeX distribution. You will also need the "swiss-army knife" of file conversion, pandoc. Pandoc is not *needed* for creating LaTeX-formatted pdfs, but if you have documents with unsupported characters and you need to use a different `pdf-engine` you'll need to use pandoc.

During the installation process, LaTeX should have been added automatically to the PATH. To test this, enter `pdflatex` into a command line/terminal. If you get the output `This is pdfTeX ...` then you are good to go. If not, please add the executable to the PATH as listed above. The executable location can be found by opening the MiKTeX/MacTeX console and looking at the bin directory under settings. For me, on a Windows computer where I don't have administrator privileges it reads `C:\Users\owner\AppData\Local\Programs\MiKTeX 2.9\miktex/bin/x64`. If you are still having issues, please consult this document.

Now you're ready to create a LaTeX-formatted pdf document. All you need to do is click *File -> Download as -> PDF via LaTeX (.pdf).*

Unfortunately, when creating pdf documents, code is not hard wrapped. This means that if you have a very long line of code (> 80 characters), it will run out of the formatted area, and at worst, off the page. At present, I do not know a way to force Jupyter notebooks to wrap the output automatically, so instead you have to write clean code and start new lines using `,` and `\n` if it's too long.

If you would rather not produce a pdf via LaTeX, instead wanting an arguably more readable output, you could create an html file. This is done in the same manner as pdf documents.

Tidy data

What do I mean by tidy data? - tidyverse info

Additional resources

This document only touches on enough to get you up an running with reproducible work. However, to become fully proficient you will need to delve deeper into the material - trust me, it'll make your life easier in the long run. Here are a few places to start for each section, many of which were the basis for the systems I implement and advocate for here.

Project structure

https://nicercode.github.io/blog/2013-04-05-projects/

https://tomwallis.info/2014/01/16/setting-up-a-project-directory/

> These two resources describe the project structure that I advocate for in a little more detail, with a few minor differences. If you work in `R` I would recommend that you follow nicercode's other suggestions as well, particularly regarding the creation of an R project.

https://medium.freecodecamp.org/why-you-need-python-environments-and-how-to-manage-them-with-conda-85f155f4353c

> This takes a deeper look into how to manage `python` environments with anaconda, and how this affects your project structures. This is useful if you are work with `python 2.x` and `python 3.x`, but also allows your to ensure old code won't get broken when modules are updated as each module is specific to the environment it is downloaded in.

Git

https://happygitwithr.com/

> I cannot emphasise this enough: this is genuinely **the best resource** I have come across for explaining how to set up and organise a project with `git`. Whilst it is aimed at `R` users, there is a large amount of cross-over, so read it regardless of the language you use.

https://medium.freecodecamp.org/how-not-to-be-afraid-of-git-anymore-fe1da7415286

> This helps you understand the nuts-and-bolts of `git` by learning to use the command line, rather than an application like SourceTree.

https://git-scm.com/book/en/v2/

> The literal book on git. Everything from the basics to the advanced.

https://nvie.com/posts/a-successful-git-branching-model/

> If you feel comfortable with the idea of branching and are interested in a good extension of what we've covered, this will help. Roughly speaking, the more complex you project is and the more people that are involved simultaneously, the more branches you want so you can handle problems as they come up, without breaking previously 'good' code.