Best practices for computational research

Nancy Chen

2/28/2019

All data analyses should be *reproducible*, *i.e.*, someone else should be able to replicate your results from your code and data. Most journals require authors to make all data and code publicly available with publication. Here are some guidelines for performing reproducible research in computational biology. Happy coding!

Keep a detailed lab notebook

Keeping a good lab notebook is crucial for computational analyses. You want to carefully document why you're performing a particular analysis as well as keep track of both what works and what does not work. You will end up redoing your analyses multiple times, so keeping good, organized notes will make your life easier. I document nearly *every* command I type.

One advantage of having a digital lab notebook is that your notes are easily searchable. For each project, keep detailed notes in chronological order in a R markdown (preferred format), LaTeX, or flat text file. Make a single, regularly-updated Table of Contents or README file in the top directory of the project to help you organize and locate all the files associated with the project.

For each analysis, include:

- **Date:** keep notes in chronological order and date everything.
- **Goal:** briefly state the overall goal of your analysis to help keep track of why you're doing the work and make searching your notes easier.
- **Approach:** outline your logic and include the actual code or list relevant scripts and input files. Note the computer you are using (personal laptop, cluster, etc.) and the directories of your files.
- **Conclusion:** interpret the results of the analysis (including any figures), list output files generated by the analysis and their locations and formats, and outline next steps. If the approach didn't work, include any error messages and discuss why you think things didn't work.

Stay organized

Keep all files (data, code, results) for a project in a single directory. Within the overall project directory, create subdirectories for raw data, code, output files, etc.

Naming files

- It helps to give your files and directories informative names that give you some idea of the contents. For example, `indivGenContribOutput.txt` is much more useful than simply `output.txt`.
- Including the date helps link to your notes and serves as version control.
- Use only alphanumeric characters and underscores in your file names, as spaces and other special characters can be annoying in Unix/Linux systems.
- Try to keep the format of your code and file names consistent, *e.g.*, don't switch between `28Feb2019CamelCaseNaming` and `pothole_case_naming_02282019`.

Writing readable code

- Please thoroughly comment your code! For each function or major chunk of code, provide the overall purpose, any assumptions, and summaries of proper usage and expected results. You want to make sure that someone else can easily understand what your code is doing and how to run it.
- Use informative names for your variables and functions.
- Use relative paths (`../rawData/FSJgenome.fasta`) instead of absolute paths (`/Users/nancy/project/rawData/FSJgenome.fasta`) so others can run your code without manually changing all the directory/file names.

Use version control

Version control software like git record changes to files and allow you to retrieve previous versions at a later time. Version control allows multiple collaborators to work simultaneously on the same set of files and can help with tracking down bugs or reverting to a previous version of the code. GitHub is an online hosting service for git repositories (a set of files; essentially the git name for projects). Storing your work in a remote git repository helps you and your collaborators stay synced and serves as an additional backup.

A very brief introduction to git

There are several good git tutorials online (*e.g.*, here's one by Karl Broman, a more detailed RStudio-focused tutorial by Jenny Bryan, and a workshop by Software Carpentry), so I'm just going to include the very basics on how to git started here.

1. Get a GitHub account. Install git, either by downloading it or using the package manager Homebrew.
2. Set up git with your user name and email. In Terminal, type

   ```
   git config --global user.name "Your name here"
   git config --global user.email "your_email@example.com"
   ```

   If you want colored output in your Terminal, type

   ```
   git config --global color.ui true
   ```

3. You can connect to GitHub using either HTTPS or SSH. To avoid typing in your username and password each time (which will get annoying), I would either set up a credential helper or SSH keys.
4. To create a new git repository, navigate to the directory of your project and type

   ```
   git init
   ```

   You can later upload ("push") your local files to a remote GitHub repository. Create a new repository on the GitHub website to get the URL (it will look like `https://github.com/username/project_repo.git` or `git@github.com:username/project_repo.git`), then use these commands:

   ```
   git remote add origin [repository URL]
   git push -u origin master
   ```

   Alternatively, if a repository already exists on GitHub, you can create a working copy of that repository locally by typing

   ```
   git clone [repository URL]
   ```

5. Tracking changes in your local repository is a two step process. First, you tell git which new/modified files to track by "adding" files to what's called the staging area:

```
git add file1 file2
```

Then, you save a snapshot of your changes to the repository as a "commit":

```
git commit -m "Short message indicating what changed"
```

6. When you have a batch of changes you're ready to upload to your remote repository on GitHub, type

   ```
   git push
   ```

   To update your local repository to the newest commit from GitHub, type

   ```
   git pull
   ```

7. To check if the repository is up to date, type

   ```
   git status
   ```

   You can check the history of all commits to a repository using

   ```
   git log
   ```

   You can see what changes you made using

   ```
   git diff
   ```

There are many more commands for creating separate branches for parallel code development, merging different commits, reverting to old commits, etc. - you can get quite fancy with git - and I recommend working through a git tutorial to learn more.

If you are uncomfortable using the command line, you can download and use a GitHub client such as Sourcetree. You can also use git in RStudio.

Some more advice

- Avoid manually editing files. Instead, try to do all data manipulation using awk/sed or scripts.
- Use scripts to automate longer pipelines.
- Document the version of all software you use and of any publically downloaded data.
- Record intermediate results to help with debugging and to allow you to start a long analysis partway instead of from the very beginning.
- Save the data and formatting variables for each figure so it's easy to recreate the figure and/or make a small visual tweak to the plot without redoing the entire analysis.
- Save seeds for random number generators.

Additional resources

Noble WS. 2009. A Quick Guide to Organizing Computational Biology Projects. *PLOS Computational Biology* 5(7): e1000424. https://doi.org/10.1371/journal.pcbi.1000424

Sandve GK, Nekrutenko A, Taylor J, Hovig E. 2013. Ten Simple Rules for Reproducible Computational Research. *PLOS Computational Biology* 9(10): e1003285. https://doi.org/10.1371/journal.pcbi.1003285

Karl Broman's course Tools for Reproducible Research