Good Enough Practices for Scientific Computing

Greg Wilson[1,*], Jenny Bryan[2], Karen Cranston[3], Justin Kitzes[4], Lex Nederbragt[5]
1) Software Carpentry Foundation / gvwilson@software-carpentry.org
2) University of British Columbia / jenny@stat.ubc.ca
3) Duke University / karen.cranston@duke.edu
4) University of California, Berkeley / jkitzes@berkeley.edu
5) University of Oslo / lex.nederbragt@ibv.uio.no
∗ E-mail: Corresponding gvwilson@software-carpentry.org

1 Introduction

Two years ago a group of researchers involved in Software Carpentry[1] and Data Carpentry[2] wrote a paper called "Best Practices for Scientific Computing[3]" [1]. It was well received, but in retrospect it might have been misguided: many novices found its catalog of tools and techniques intimidating, and by definition, the "best" are a minority, so "best practices" are what only a minority does.

This paper therefore looks instead at "good enough" practices[4], i.e., at the minimum set of practices we believe every researcher can and should adopt. It draws inspiration from several sources [2–8], and from both our personal experiences in research computing and the experiences of the thousands of people who have taken part in Software and Data Carpentry workshops over the past six years.

Our intended audience is researchers who are working alone or with a handful of collaborators on projects lasting a few days to a few months, and who are just starting to move beyond saving spreadsheets called `results-updated-3-revised.xlsx` in Dropbox. A practice is included in our list if large numbers of *computationally competent*[5] researchers use it, and if large numbers of novices try it after a short introduction, and are still using it months after first trying it out. We use these criteria because there's no point recommending things that people don't actually use, won't try, or abandon after a few days or weeks because they don't perceive value.

That last stipulation is important. Many of our recommendations are for the benefit of the collaborator every researcher cares about most: their future self[6]. If researchers don't see those benefits quickly enough to pay for the slowdown that inevitably occurs when adopting a new tool or practice, they will almost certainly switch back to their old way of doing things. As we discuss in Section 8 this rules out many practices such as code review that we feel are essential for larger-scale development.

add pointers to papers or projects that follow these rules.

Many of these guidelines are phrased in terms of helping other people, such as current and future collaborators. In all cases, though, the most important "other person" you're helping is your future self: for example, a good to-do list will save you time when you come back to your project next year. It's always tempting to cut corners, but your past self doesn't answer email.

[1] `http://software-carpentry.org/`
[2] `http://datacarpentry.org/`
[3] `http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745`
[4] Note that English lacks a good word for this: "mediocre", "adequate", and "sufficient" aren't exactly right.
[5] We use the term "computationally competent" to mean someone who can accomplish routine tasks without heroic effort, i.e., someone who's as good at data analysis or programming as most adults are at driving a car. We use the term (a) to make it clear that we're trying to inculcate the equivalent good lab skills, rather than turn researchers into professional programmers, and (b) because the terms "computational literacy" and "computational thinking" have been used in so many ways that it's hard to know what they mean any more.
[6] As the joke goes, your past self doesn't answer email...

2 Data Management

Project data may need to exist in various forms, ranging from what first arrives to what's included in the final publication. We recommend that researchers follow six rules.

First, *save data in the rawest form available*. Where possible, save the data file produced by an instrument or raw results from a survey, with all of its mystifying imperfections (e.g., raw JPEGs for photographs). It is tempting to overwrite raw data files with cleaned-up versions, but faithful retention is essential to being able to re-run analyses from start to finish. Being able to produce the raw data also enhances confidence in your final results. More immediately, having the raw data helps allows you to recover from analytical mishaps and allows you to experiment without fear.

But use common sense: if a large volume of data is received in a format that is storage-inefficient or computationally inefficient to work with, transform it for storage with a lossless, well-documented procedure. Equally, don't duplicate contents of stable, long-lived repositories (i.e., don't clone GenBank)

You may also consider restricting file permissions to read-only, even (or especially) for yourself, so it is harder to damage raw data by accident or to hand edit it in a moment of weakness.

Second *create the data you wish to see in the world*, i.e., create the dataset you *wish* you had received from the raw data and use that as the starting point for downstream analyses (including those you didn't initially think of doing). This is the time to maximize both machine and human readability, but *not* to do vigorous data filtering nor to bring in external information.

Approach this initial tidying up as internal to the existing dataset and non-destructive, both of the data and its general "shape". Enhance machine readability by converting from proprietary or high-friction formats (e.g., Microsoft Excel or XML) to open and simple formats (e.g., comma-separated values). Enhance human readability by replacing inscrutable variable names and artificial data codes with self-explaining alternatives. For example, rename the variables `name1` and `name2` to `personal_name` and `family_name`, recode the treatment variable from `1` vs. `2` to `untreated` vs. `treated`, and replace artificial codes, such as "-99" for missing data, with proper `NA`s.

Both human and machine readability can be enhanced by storing especially useful metadata as part of the filename itself, while keeping the filename regular enough for easy pattern matching. For example, a filename like `2016-05-alaska-b.csv` makes it easy for both people and programs to select by year (`2016-*.csv`), by location (`*-alaska-*.csv`), and so on.

Where possible, use open non-proprietary formats to closed, proprietary format: they'll likely last longer. CSV is good for simple tabular data, JSON, YAML, or XML for non-tabular data such as graphs (the node-and-arc kind), and HDF5 for certain kinds of structured data. This guide[7] has a few useful guidelines for choosing a file format.

Third, *create analysis-friendly data*. You can do this by reformatting data to eliminate processing steps between loading the data set and doing the analysis.

1. Columns that contain more than one variable's worth of information should be split. For example, the presence of "kg" in "3.4 kg" will cause most analytical environments to read this in as character data rather than numeric. It should be split into two columns (the mass "3.4" and the units "kg"), or the units should be recorded in the variable name and/or metadata.

2. Multiple columns that only contain one variable's worth of information when taken together should be combined. This is characteristic of data that has been laid out for human eyeballs or for manual data entry. For example, there might be one row per field site and then columns for measurements made at each of several time points. It is convenient to store this in a "short and wide" form for data entry and inspection, but for most analyses it will be advantageous to gather these columns into a variable of measurements, accompanied by a companion variable indicating the time point.

[7] `http://www.library.illinois.edu/sc/services/data_management/file_formats.html`

3. Reformat values as needed to match your environment's built-in parsing rules. For example, use date-time format that will be recognized automatically.

And fourth, *always record the steps used to clean up data*: that's as much a part of your analysis as any statistics you might do.

The goal of this refactoring is to create "tidy data", which can be a powerful accelerator for analysis [5, 8]. This is the time to make sure your data will play nicely with your analytical environment and plans. This pass through the data likely does not change the amount of data, but may dramatically alter its form.

OpenRefine

OpenRefine[8] is an excellent tool for this stage of data cleanup. It combines a spreadsheet-like interface to tabular data with a large set of cleanup heuristics, and can generate a trace of cleanup steps to ensure reproducibility.

Fifth, *give every record a unique key.* In a well-organized data set, it's easy to select a specific record to update or delete. Using fields in the data (such as people's names or addresses) does not guarantee this; using row numbers in tables does, but those numbers will change as records are added, removed, or moved.

We therefore recommend giving each record a unique, persistent key that has no other purpose than identifying the record[9] The provision of such a key also makes it much easier to link data sets together.

Sixth, *marshal complementary data in files that meet the same standards.* Raw data frequently does not fully explain itself. For example, if each well of a microtitre plate is used to study a specific gene knockout, the prepared raw data will have a `well` variable taking on values like "A1" and "G12", but the plate reader cannot possibly know what's in each well. That information will be absent from the primary data file, so you must create a supplementary file in order to look this up. Make sure to use the same names and codes when variables in two datasets refer to the same thing to simplify merging and lookup (and make them less error-prone).

Seventh, *create a dataset purpose-built for specific analyses and figures.* This probably involves filtering rows, selecting relevant variables, and merging with external information. If the preceding steps have been done, this can often be surprisingly straightforward. The basic form of the data was hopefully set earlier, so the main changes here are likely to be data reduction and the amalgamation of multiple datasets.

Eighth and last, *submit data to a reputable DOI-issuing repository so that others can access and cite it.* Your data is as much a product of your research as the papers you write, and just as likely to be useful to others (if not more so).

Discussion

The data processing strategy advocated above is divided into steps and produces intermediate data files, with increasing levels of cleanliness and task-specificity. While there is growing appreciation for reproducibility—i.e., being able to re-run an analysis start to finish—it is also extremely useful to be able to re-run *parts* of a pipeline. The reasons are similar to those given for modularity in computer code: by breaking data preparation into steps, it becomes easier to revisit later and to only tinker with specific data cleaning operations.

We propose delimited plain text as an appealing "lowest common denominator" data form that offers high usability across time, people, operating systems, and analytical environments. We recognize, though,

[9] Student numbers and similar identifiers serve this purpose in institutional databases.

that plain text is not a panacea. In particular, when collaborating with others, be aware of and, when possible, standardize on file encoding, line endings, and the elimination of prose-oriented features, such as "smart quotes". Differences between collaborators in these pesky details can cause the changes between file versions to be large and therefore substantially less comprehensible.

Choosing Metadata

Elizabeth Wickes[10] provides a useful classification for metadata, with implications for it how it should be represented. She notes it is easy to conflate metadata about the dataset as a whole with metadata about the content, e.g., individual columns. Most metadata schemas are aimed at the former, i.e., they detail the author, funder, related publications, etc. When considering how to store metadata, consider the intended audience. Is it humans? Write a README. Is it machines, such as metadata harvesters and formal repositories? Create an impeccably formatted metadata file.

3 Software

If you or your group are creating tens of thousands of lines of software which is going to be used by hundreds of people you have never met, then you are doing software engineering. If you're writing a few pages now and again, and are probably going to be its only user, you can still make things easier on yourself by adopting a few key practices. What's more, adopting these practices will make it easy for people to understand and (re)use your code, which in turn makes it more likely that they collaborate with you and/or give you credit for your work.

The core realization in these practices is that *readable*, *reusable*, and *testable* are not separate things. They are all side effects of *modular* code, i.e., of building programs out of short, single-purpose functions with clearly-defined inputs and outputs. Building software this way is the key to both productivity and reproducibility.

The first recommendation is that *every analysis step should be represented textually* (complete with parameter values). It isn't always possible to store steps as text (e.g., manual selection of region of interest in image), but it is always possible to store the result for later checking.

Second, *place a brief explanatory comment at the start of every program*, no matter how short it is. That comment should include at least one example of how the program is used: remember, an example is worth a thousand words. Where possible, the comment should also indicate reasonable values for parameters. For example:

```
Synthesize image files for testing circularity estimation algorithm.

Usage: generate_images.py -d p_diameter -f p_flaws -r rand_seed  -s image_size

where:
-f fuzzing     = fuzzing range of blobs (typically 0.0-0.2)
-n p_flaws     = p(success) for geometric distribution of # flaws/sample (typically 0.5-0.8)
-o output_file = name of output file
-r p_radius    = p(success) for geometric distribution of flaw radius (typically 0.1-0.4)
-s rand_seed   = RNG seed (large integer)
-v             = verbose
-w size        = image width/height in pixels (typically 480-800)
```

Third, *decompose programs into functions* that:

- are no more than one page long (60 lines, including spaces and comments),

- do not use global variables (constants are OK), and
- take no more than half a dozen parameters.

The key motivation here is to fit the program into the most limited memory of all: ours. Human short-term memory is famously incapable of holding more than about seven (plus or minus two) items at once. If we are to understand what our software is doing, we must break it into chunks that obey this limit, then create programs by composing chunks into larger chunks and so on.

Orwell's Sixth Rule

George Orwell's essay "Politics and the English Language" presents six rules for writing well [9]. The sixth and final states, "Break any of these rules sooner than say anything outright barbarous," which is good advice for writing software as well.

Fourth, *be ruthless about eliminating duplication.* Write and re-use functions instead of copying and pasting source code, and use data structures like lists rather than creating lots of variables called `score1`, `score2`, `score3`, etc.

This applies with even greater force to libraries. The easiest code to debug and maintain is code that you didn't actually write, so *always search for libraries that do what you need* before writing new code yourself.

Fifth, *give functions and variables meaningful names*, both to document their purpose and to make the program easier to read. As a rule of thumb, the greater the scope of a variable, the more informative its name should be: while it's acceptable to call the counter variable in a loop `i` or `j`, the main grid for your simulation should *not* have a one-letter name.

Sixth, *make dependencies and requirements explicit.* This is usually done on a per-project rather than per-program basis, i.e., by adding a file called something like `requirements.txt` to the root directory of the project, or by adding a "Getting Started" section to the `README` file. (More sophisticated users may structure the requirements in a way that installation management tools can use.)

Seventh, *do not comment and uncomment sections of code to control a program's behavior*, since this is error prone and makes it difficult or impossible to automate analyses. Instead, put if/else statements in the program to control what it does.

Logging

One special case of this recommendation is to use a logging library such as Java's `log4j` for debugging messages. When this is done, messages are left in the code, but their activity is controlled by an external configuration file.

Eighth, *provide a simple example or test data set* that users (including yourself) can run to determine whether the program is working at all and whether it gives a known correct output for a simple known input. Such as "build and smoke test" is particularly helpful when supposedly-innocent changes are being made to the program, or when it has to run on several different machines, e.g., the developer's laptop and the department's cluster.

Ninth and last, *submit code to a reputable DOI-issuing repository* upon submission of paper, just as you do with data. Your software is as much a product of your research as your papers, and should be as easy for people to credit.

4 Collaboration

You may start working on your project by yourself or with a small group of collaborators whom you now. But even at an early stage, you may want to open your project so that you can attract potential new collaborators whom you *don't* already know. Even if you choose not to do this at the start of the project, following a few simple rules will make it easier to do so later.

To enhance collaboration, aim for:

1. *Simplicity:* the easier it is for people to collaborate, the more likely they are to do so (and to give you credit)
2. *Low entry:* Remove the two most reported barriers to contributing from Steinmacher et al[11]:
    - finding a task to start on
    - setting up the local workspace to start work
3. *Clarity:* remove uncertainty around what a potential collaborator is allowed to do

First, *create an overview of your project.* Have a short `README` file explaining the project's purpose in an easy-to-find place (such as the project's home directory). This file is often the first thing users of your project will look at, so make it explicit already here that you welcome contributors and point them to the ways to help out.

The `README` file (which may be called that, or `README.txt`, or something similar) should contain:

- the project's title,
- a brief description (similar to the abstract of a paper), and
- contact information that actually works.

You should also create a `CONTRIBUTING` file that describes what people need to do in order to get the project going and contribute to it: dependencies that need to be installed, tests that can be run to ensure the software has been installed correctly, and guidelines or rules that your project adheres to (e.g., checklists you may use before accepting a suggested change).

Second, *create a shared public "to-do" list.* This could be a plain text file containing the to-do list called `notes.txt` or `todo.txt` or something similar. Alternatively, you could use the *issue* functionality on sites such as GitHub or Bitbucket to create a new issue for each to-do item. (You can even add labels such as "low hanging fruit" to point newcomers at issues that are good starting points.) Whatever your system of choice, make the descriptions of the items clear enough so that they make sense for new collaborators.

Third, *make the license explicit.* Have a `LICENSE` file in the project's home directory that clearly states what license(s) apply to the project's software, data, and manuscripts. Lack of an explicit license does not mean there isn't one; rather, it implies the author is keeping all rights and others are not allowed to re-use or modify the material. We recommend:

- Use Creative Commons licenses for data and text, either CC-0[12], the "No Rights Reserved" license, or CC-BY[13], the "Attribution" license, allowing sharing and reuse but requiring giving appropriate credit to the creator(s).

[11] `http://lapessc.ime.usp.br/work.jsf?p1=15673`
[12] `https://creativecommons.org/about/cc0/`
[13] `https://creativecommons.org/licenses/by/4.0/`

- Use a permissive license (e.g., the MIT, BSD, or Apache[14] licenses) for software.

We recommend *against* the "no commercial use" variations of the Creative Commons licenses because they may impede some forms of re-use. For example, if a researcher in a developing country is being paid by her government to compile a public health report, and wishes to include some of your data, she will be unable to do so if the license says "non-commercial". We recommend permissive software licenses rather than the GNU General Public License[15] (GPL) because it is easier to integrate permissively-licensed software into other projects. (Note that it is straightforward to switch from a permissive license to the GPL if you should change your mind later but rather more complicated to go in the other direction.)

Fourth, *make the project citable* by including a `CITATION` file in the project's home directory that describes:

- how to cite this project as a whole, and
- where to find, and how to cite, any data sets, code, figures, and other artifacts that have their own DOIs.

For example, the `CITATION` file for the Software Carpentry project reads:

```
To reference Software Carpentry in publications, please cite:

Greg Wilson: "Software Carpentry: Lessons Learned". F1000Research,
2016, 3:62 (doi: 10.12688/f1000research.3-62.v2).

@online{wilson-software-carpentry-2016,
  author      = {Greg Wilson},
  title       = {Software Carpentry: Lessons Learned},
  version     = {2},
  date        = {2016-01-28},
  url         = {http://f1000research.com/articles/3-62/v2},
  doi         = {10.12688/f1000research.3-62.v2}
}
```

For an example of a more detailed `CITATION` file, see the one for the khmer project[16].

Discussion

The overview and to-do list are to help you as well as other people—remember, your most important collaborator is yourself six months from now. The license and citation file are there to make it easy for other people to help you and give you credit for your work.

5 Project Organization

The previous sections have described practices for creating, naming, and storing the files that make up a research project. Organizing these files in a logical and consistent directory structure will help you keep track of them and help others review your work more easily.

Our recommendations are drawn from [3], with a healthy admixture from [2]. Each project should be in its own directory, which should be named after the project, and should contain the following subdirectories:

[14] `https://www.safaribooksonline.com/library/view/understanding-open-source/0596005814/ch02.html`
[15] `https://www.safaribooksonline.com/library/view/understanding-open-source/0596005814/ch03.html`
[16] `https://github.com/dib-lab/khmer/blob/master/CITATION`

1. `doc` contains text documents associated with the project. This may include files for manuscripts, documentation for source code, and/or an electronic lab notebook recording your experiments. Subdirectories may be created for these different classes of files.

2. `data` contains raw data and metadata, organized into subdirectories if needed. Note that this directory contains only unprocessed data: cleaned or otherwise modified data files are treated like results.

3. `src` contains the source code for scripts and programs, both that written in interpreted languages such as R or Python and that written compiled languages like Fortran, C++, or Java. Shell scripts, snippets of SQL used to pull information from databases, and everything else needed to regenerate the results are all managed like source code.

4. `bin` contains executable scripts and programs that are brought in from other sources or compiled from code in the `src` directory. Projects that use only interpreted languages, such as R, Python, or the Unix shell, will not require this directory.

5. `results` for all files that are generated as part of the project. This includes both intermediate results, such as cleaned data sets or simulated data, as well as final results such as figures and tables.

All files should be named clearly to reflect their contents (e.g., `bird_count_table.csv`, `manuscript.md`) or their functionality (e.g., `sightings_analysis.py`), *not* using sequential numbers (e.g., `result1.csv`, `result2.csv`) or a location in a final manuscript (e.g., `fig_3_a.png`), since those numbers will almost certainly change as the project evolves.

What's a "Project"?

Like deciding when a chunk of code should be made a function, the ultimate goal of dividing research into distinct projects is to help you and others best understand your work. Some researchers create a separate project for each manuscript they are working on, while others group all research on a common theme, data set, or algorithm into a single project.

As a rule of thumb, divide work into projects based on the overlap in data and code files. If two research efforts share no data or code, they will probably be easiest to manage independently. If they share more than half of their data and code, they are probably best managed together. Anything in between can be decided based on the set of people you're collaborating with.

The `src` directory often contains two conceptually distinct types of files that should be distinguished either by clear file names or by additional subdirectories. The first type are individual files or related groups of files that contain functions to perform the core analysis of the research. One file, for example, may contain functions used for data cleaning, while another contains functions to do certain statistical analyses. These files can be thought of as the "scientific guts" of the project, and as the project grows, they can be organized into additional subdirectories. If a project were to include re-runnable tests (such as unit tests or regression tests—see Section 8), these are the functions that would be tested.

The second type of file in the `src` directory is controller or driver scripts that combine the core analytical functions with particular parameters and data input/output commands in order to execute the entire project analysis from start to finish. A controller script for a simple project, for example, may read a raw data table, import and apply several cleanup and analysis functions from the other files in this directory, and create and save a numeric result. For a small project with one main output, a single controller script should be placed in the main `src` directory and distinguished clearly by a name such as "runall".

The controller script is the glue that holds the analysis together and allows a single command, such as `python runall.py`, to re-run the entire analysis from start to finish. These scripts should be short (no more than 100-200 lines at most), and be very easy to understand: in particular, while they may contain loops (to process multiple data files or sweep across parameter ranges) they should contain few if any conditional statements or new function definitions. If a control script becomes longer or more complicated than this, or begins to include code that would require a new collaborator more than a minute or two to understand, those portions of the code should be moved out of the controller script and into other core analysis files in this directory.

The `results` directory will generally require additional subdirectories for all but the simplest projects. Intermediate files such as cleaned data, statistical tables, and final publication-ready figures or tables should be separated clearly by file naming conventions or placed into different subdirectories; those belonging to different papers or other publications should be grouped together.

The figure below provides a concrete example of how a simple project might be organized following these recommendations. The root directory contains a `README` file that provides an overview of the project as a whole and a `CITATION` file that explains how to reference it. The `data` directory contains a single CSV file with tabular data on bird counts (machine-readable metadata could also be included here). The `src` directory contains `sightings_analysis.py`, a Python file containing functions to summarize the tabular data, and a controller script `runall.py` that loads the data table, applies functions imported from `sightings_analysis.py`, and saves a table of summarized results in the `results` directory.

```
.
|-- CITATION
|-- README
|-- data/
|   \-- birds_count_table.csv
|-- doc/
|   |-- notebook.md
|   \-- manuscript.md
|-- results/
|   \-- summarized_results.csv
|-- src/
|   |-- sightings_analysis.py
|   \-- runall.py
```

This project doesn't have a `bin` directory, since it does not rely on any compiled software. The `doc` directory contains two text files written in Markdown, one containing a running lab notebook describing various ideas for the project and how these were implemented and the other containing a running draft of a manuscript describing the project findings.

6 Version Control

Keeping track of changes that you or your collaborators make to data, software, and manuscripts is a critical part of research. Version control systems were developed to do exactly this for software: they keep track of what was changed in a file when and by whom, and usually synchronize changes to a central server so that many users can track the same set of files.

We believe that the best tools for providing these capabilities are version control systems that are widely used in software development, such as Git, Mercurial, and Subversion (which all provide the same basic functionality). However, many newcomers find these tools difficult to learn, and their benefits often only become apparent on projects that are larger and more complex than the "solo researcher" projects common in many academic fields.

For those not yet ready to adopt version control, we recommend a systematic manual approach for managing changes that can be appropriate for small projects with one or a few investigators working closely together.

Whatever approach is adopted should help with the following:

1. It should aid *reproducibility* by allowing you to reference or retrieve a specific version of the entire project. This is valuable for your future self (when you finally get the reviews back for your paper), for your lab-mates and collaborators (in case you leave the project), and for reviewers, editors, and others who want to convince themselves of the conclusions in your published research. It also helps your future self come back to a project after a long field season and *not* have to spend days remembering what you did last or figuring out what your collaborators have changed while you have been gone.

2. It should aid *fixability*: having access to every version of code, figures, and data helps you figure out why Figure 4 looks different now than it did last week. To aid this, it should also provide an automatic and systematic way to identify particular versions.

3. It should support *sharing and collaboration*, particularly by managing the process of merging independent changes made by different people, and distributing those changes back to everyone in a controlled, traceable way.

Our first suggested approach, in which everything is done by hand, has three parts. First, *store each project in a folder that is mirrored off the researcher's working machine* by a system such as Dropbox, and synchronize that folder at least daily. It may take a few minutes, but that time can be spent catching up on email, and is repaid the moment a laptop is stolen or its hard drive fails.

Second, *add a file called* `CHANGELOG.txt` *to the project's* `docs` *subfolder*, and make dated notes about changes to the project in this file in reverse chronological order (i.e., most recent first). An example of such a file is:

```
## 2016-04-08

* Switched to cubic interpolation as default.
* Moved question about family's TB history to end of questionnaire.

## 2016-04-06

* Added option for cubic interpolation.
* Removed question about staph exposure (can be inferred from blood test results).
```

Third, *copy the entire project whenever a significant change has been made* (i.e., one that materially affects the results being produced), and store that copy in a sub-folder whose name reflects the date in the area that's being synchronized. This approach results in projects being organized like this:

```
.
|-- project_name/
|   \-- current
|       \-- ...project content as described earlier...
|   \-- 2016-03-01
|       \-- ...content of 'current' on Mar 1, 2016
|   \-- 2016-02-19
|       \-- ...content of 'current' on Feb 19, 2016
```

Here, the `project_name` folder is mapped to external storage (such as Dropbox), `current` is where development is done, and other folders within `project_name` are old versions.

Coyping everything like this may seem wasteful, since many files won't have changed, but consider: a terabyte hard drive costs about $50 retail, which means that 50 GByte costs less than a latte. Provided large data files are kept out of the backed-up area (discussed below), this approach costs less than the time it would take to select files by hand for copying.

This manual procedure meets the core requirements outlined above without needing any new tools. If multiple researchers are working on the same project, though, they will need to coordinate so that only a single person is working on specific files at any time. In particular, they may wish to create one change log file per contributor, and to merge those files whenever a backup copy is made.

What the manual process described above requires most is self-discipline. The version control tools that underpin our second approach—the one we all now use for our projects—don't just accelerate the manual process: they also automate some steps while enforcing others, and thereby require less self-discipline for more reliable results.

A version control system stores snapshots of a project's files in a repository. Users can modify their working copy of the project at will, and then commit changes to the repository when they wish to make a permanent record and/or share their work with colleagues. The version control system automatically records when the change was made and by whom along with the changes themselves.

Crucially, if several people have edited files simultaneously, the version control system will detect the collision and require them to resolve any conflicts before recording the changes. Modern version control systems also allow repositories to be synchronized with each other, so that no one repository becomes a single point of failure. Together, these features have several benefits:

1. Instead of requiring users to copy everything into subfolders, version control safely stores just enough information to allow old versions of files to be re-created on demand. This saves both space and time.

2. Instead of relying on users to choose sensible names for backup copies, the version control system timestamps all saved changes automatically.

3. Instead of requiring users to be disciplined about writing log comments, version control systems prompt them every time a change is saved. They also keep a 100% accurate record of what was *actually* changed, as opposed to what the user *thought* they changed, which can be invaluable when problems crop up later.

4. Instead of simply copying files to remote storage, version control checks to see whether doing that would overwrite anyone else's work. This turns out to be the key to supporting large-scale ad hoc collaboration.

Discussion

This section was easily the most contentious in this paper. On the one hand, all of the authors use version control daily for all of their projects. On the other hand, many competent and productive researchers who have used version control in earnest believe that using it for managing papers and data is like using the back end of a screwdriver to hammer in a nail. As Arjun Raj said in a blog post:

> Google Docs excels at easy sharing, collaboration, simultaneous editing, commenting and reply-to-commenting. Sure, one can approximate these using text-based systems and version control. The question is why anyone would like to...
>
> The goal of reproducible research is to make sure one can... reproduce... computational analyses. The goal of version control is to track changes to source code. These are fundamentally

> distinct goals, and while there is some overlap, version control is merely a tool to help achieve that, and comes with so much overhead and baggage that it is often not worth the effort.
>
> I've heard some say some version of "If you're not using version control, it's not science." This assertion is of course patently false, as plenty of people have been doing fine science without version control. Widespread adoption will require practical, non-ideological reasons why one should use version control, and some consideration of the serious hassles associated with its use.

Regardless of which approach is taken, the first rule for using version control is that *almost everything created by a human being should be backed up as soon as it is created.* This includes scripts and programs of all kinds, software packages that your project depends on, and documentation. A few exceptions to this rule are discussed below.

Second, *make changes small.* Each change committed to a version control system or added to a change log should not be so large as to make the change tracking irrelevant. For example, a single change such as "Revise script file" that adds or changes several hundred lines is likely too large, as it will not allow changes to different components of an analysis to be investigated separately. Similarly, changes should not be broken up into pieces that are too small, although we find that this is less of a danger with novices. As a rule of thumb, a good size for a single change is a group of edits that you could imagine wanting to undo in one step at some point in the future.

Third, everyone working on the project should *share changes frequently*, and incorporate changes from others just as frequently. Do not allow individual investigator's versions of the project repository to drift apart, as the effort required to merge differences goes up faster than the size of the difference. This is particularly important for the manual version control procedure describe above, which does not provide any assistance for merging simultaneous changes.

Finally, *create a checklist for committing and sharing changes* to the project, document the checklist in the repository, and *use it.* The list might include:

- writing commit messages that clearly explain any changes,
- size and content of single commits,
- style guidelines for code,
- updating to-do lists, and
- bans on committing half-done work or broken code.

When is Version Control Not Necessary?

Despite the benefits of version control systems, there are some types of files that may *not* be appropriate for version control. First, data should be backed up, but may or may not be a good candidate for version control[17]. The size of data sets can be a problem. If a file is small, placing it in a version control repository facilitates reproducibility. On the other hand, today's version control systems are not designed to handle megabyte-sized files, never mind gigabytes, though support for them is emerging.

Raw data should not change (and therefore should not require version tracking). Putting synthesized or modified data sets into version control may not be necessary if you can re-generate these files from raw data and data-cleaning scripts (which definitely *are* under version control!).

Unfortunately, some data formats are not amenable to version control, which is designed to work with text files where the lines are in a specific order (such as source code or plain-text documentation).

[17] We also believe that data should be made publicly available except when there are strong reasons not to do so, such as patient confidentiality, but that is a separate issue.

Binary files (such as HDF5) can be put in a version control system, but you won't be able to see specific changes, although GitHub has some clever tools for monitoring changes in image files. Similarly, tabular data (such as CSV files) can be put in version control, but changing the order of the rows or columns will create a big change for the version control system, even if the data itself has not changed.

Also, researchers dealing with data subject to legal restrictions that prohibit sharing (such as medical data) should be careful not to put data in public version control systems. Some institutions may provide access to private version control systems, so it is worth checking with your IT department.

Second, opinion is divided on whether the `results` directory and other generated files such as figures should be placed under version control. If we borrow conventions from software development (just as we borrowed version control itself) the answer is no, but there are some benefits to putting results under version control in data analysis projects:

1. It gives collaborators immediate access to current processed data, results, figures, etc., without needing to regenerate it all.

2. Version control facilitates *diffing*, i.e., seeing the differences between old and new states of files. Diffs can be used to see the downstream effects of actions like upgrading a piece of software, refactoring a script, or starting with a slightly different dataset.

If results files are kilobytes or a few megabytes in size, we therefore recommend keeping them under version control. Anything more than this, and something else should be used for management.

7 Manuscripts

Gathering data, analyzing it, and figuring out what it means is the first 90% of any project; writing up is the other 90%. While the writing step is rarely addressed in discussions of scientific computing, computing has changed the process of manuscript generation just as much as it has changed the research itself.

A common practice is for the lead author to send successive versions of a manuscript to coauthors to collect feedback, which is returned as changes to the document, comments on the document, plain text in email, or a mix of all three. The result is a lot of files to keep track of and much tedious manual labor to merge all the comments to create a new master version, at which point the process begins again.

Instead of an email-based workflow, we recommend mirroring good practices for managing software and data, to make writing scalable, collaborative, and reproducible. As with our recommendations for version control in general, we suggest that groups choose one of two different approaches for managing manuscripts. The goals of both alternatives are to:

1. Ensure that text is accessible to yourself and others now and in the future by making a single master document that is available to all coauthors at all times.

2. Reduce the chances of work being lost or people overwriting each other's work.

3. Make it easy to track and combine contributions from multiple collaborators.

4. Avoid duplication and manual entry of information, particularly in constructing bibliographies, tables of contents, and lists.

5. Make it easy to regenerate the final published form (e.g., a PDF) and to tell if it is up to date.

6. Make it easy to share that final version with collaborators and to submit it to a journal.

The First Rule Is...

Which workflow is chosen is less important than having all authors agree on one or the other *before* writing starts. Make sure to also agree on a single method to provide feedback, be it an email thread or mailing list, an issue tracker (like the ones provided by GitHub and Bitbucket), or some sort of shared online to-do list.

Our first recommended approach has two parts. First, *write manuscripts using online tools with rich formatting, change tracking, and reference management*, such as Google Docs. Second, *include a `PUBLICATIONS` file in the project's `doc` directory* containing metadata about each online manuscript (e.g., their URLs). This is analogous to the `data` directory, which might contain links to the location of the data file(s) rather than the actual files.

We realize that in many cases, even this solution is asking too much from those who see no reason to move forward from desktop GUI tools. To satisfy them, the manuscript can be converted to a desktop editor file format (e.g., Microsoft Word's `.docx` or LibreOffice's `.odt`) after major changes, then downloaded and saved in the `doc` folder. Unfortunately, this means merging some changes and suggestions manually, as existing tools cannot always do this automatically when switching from a desktop file format to text and back (although `[pandoc]` can go a long way).

The second approach treats papers exactly like software, and has been used by researchers in mathematics, astronomy, physics, and related disciplines for decades.

First, *write the manuscript in a plain text format that permits version control* such as LaTeX[18] or Markdown[19], and then convert them to other formats such as PDF as needed using scriptable tools like Pandoc[20]. Second, *include tools needed to compile manuscripts in the project folder* and keep them under version control just like tools used to do simulation or analysis.

This approach re-uses the version control tools and skills used to manage data and software, and is a good starting point for fully-reproducible research. However, it requires all contributors to understand a much larger set of tools, including markdown or LaTeX, make, BiBTeX, and Git/GitHub.

Discussion

An early draft of this paper recommended plain text in version control as the best way to manage manuscripts, but several reviewers pushed back forcefully. For example, Stephen Turner wrote:

> ...try to explain the notion of compiling a document to an overworked physician you collaborate with. Oh, but before that, you have to explain the difference between plain text and word processing. And text editors. And markdown/LaTeX compilers. And BiBTeX. And Git. And GitHub. Etc. Meanwhile he/she is getting paged from the OR...
>
> ...as much as we want to convince ourselves otherwise, when you have to collaborate with those outside the scientific computing bubble, the barrier to collaborating on papers in this framework is simply too high to overcome. Good intentions aside, it always comes down to "just give me a Word document with tracked changes," or similar.

In keeping with our goal of recommending "good enough" practices, we have therefore included online storage in something like Google Docs. We still recommend *against* traditional desktop tools like LibreOffice and Microsoft Word because they make collaboration more difficult than necessary:

- If the document lives online (e.g., in Google Docs) then everyone's changes are in one place, and hence don't need to be merged manually.

[18] `http://www.latex-project.org/`
[19] `http://daringfireball.net/projects/markdown/`
[20] `http://pandoc.org/`

- If the document lives in a version control system, it provides good support for finding and merging differences resulting from concurrent changes. It also provides a convenient platform for making comments and performing review.
- Both of our recommendations clearly define the master document and allow everyone to contribute to it on an equal footing.

As an example of an implementation of these recommendations, this paper was written using a central online repository (GitHub), the *issue* functionality for discussing the outline and text, and *pull requests* for reviewing the contributions from different authors, including collecting comments and suggestions on them, and other contributors before merging them in.

Supplementary Materials

Supplementary materials often contain much of the work that went into the project, such as tables and figures or more elaborate descriptions of the algorithms, software, methods, and analyses. In order to make these materials as accessible to others as possible, do not rely solely on the PDF format, since extracting data from PDFs is notoriously hard. Instead, we recommend separating the results that you may expect others to reuse (e.g., data in tables, data behind figures) into separate, text-format files. The same holds for any commands or code you want to include as supplementary material: use the format that most easily enables reuse (source code files, Unix shell scripts etc).

8 What's Not on This List

We have deliberately left many good tools and practices off our list, including some that we use daily.

Branches A *branch* is a "parallel universe" within a version control repository. Developers create branches so that they can make multiple changes to a project independently. They are central to the way that experienced developers use systems like Git, but they add an extra layer of complexity to version control for newcomers. Programmers got along fine in the days of CVS and Subversion without relying heavily on branching, and branching can be adopted without significant disruption after people have mastered a basic edit-commit workflow.

Build Tools Tools like Make[21] were originally developed to recompile pieces of software that had fallen out of date. They are now used to regenerate data and entire papers: when one or more raw input files change, Make can automatically re-run those parts of the analysis that are affected, regenerate tables and plots, and then regenerate the human-readable PDF that depends on them. However, novices can achieve the same behavior by writing shell scripts that re-run everything; these may do unnecessary work, but given the speed of today's machines, that is unimportant for small projects.

Unit Tests A *unit test* is a small test of one particular feature of a piece of software. Projects rely on unit tests to prevent *regression*, i.e., to ensure that a change to one part of the software doesn't break other parts. While unit tests are essential to the health of large libraries and programs, we have found that they usually aren't compelling for solo exploratory work. (Note, for example, the lack of a `test` directory in Noble's rules [3].) Rather than advocating something which people are unlikely to adopt, we have left unit testing off this list.

Continuous Integration Tools like Travis-CI[22] automatically run a set of user-defined commands whenever changes are made to a version control repository. These commands typically execute

[21] `https://www.gnu.org/software/make/`
[22] `https://travis-ci.org/`

tests to make sure that software hasn't regressed, i.e., that things which used to work still do. These tests can be run either before the commit takes place (in which case the changes can be rejected if something fails) or after (in which case the project's contributors can be notified of the breakage). CI systems are invaluable in large projects with many contributors, but pay fewer dividends in smaller projects where code is being written to do specific analyses.

Profiling and Performance Tuning *Profiling* is the act of measuring where a program spends its time, and is an essential first step in *tuning* the program (i.e., making it run faster). Both are worth doing, but only when the program's performance is actually a bottleneck: in our experience, most users spend more time getting the program right in the first place.

Coverage Every modern programming language comes with tools to report the *coverage* of a set of test cases, i.e., the set of lines that are and aren't actually executed when those tests are run. Mature projects run these tools periodically to find code that isn't being used any more, but as with unit testing, this only starts to pay off once the project grows larger, and is therefore not recommended here.

The Semantic Web Ontologies and other formal definitions of data are useful, but in our experience, even simplified things like Dublin Core[23] are rarely encountered in the wild.

Data Structures Associative data structures (e.g., Python's dictionaries), graphs (of the node-and-arc kind), Bloom filters, and many other data structures are essential to making programs both correct and efficient, but there are too many, and they are too specific to particular problems, for us to single any out for this list.

Documentation Good documentation is a key factor in software adoption, but in practice, people won't write comprehensive documentation until they have collaborators who will use it. They will, however, quickly see the point of a brief explanatory comment at the start of each script, so we have recommended that as a first step.

A Bibliography Manager Researchers should use a reference manager of some sort, such as Zotero[24], and should also obtain and use an ORCID[25] to identify themselves in their publications, but discussion of those is outside the scope of this paper.

Code Reviews and Pair Programming These practices are valuable in projects with multiple contributors, but are hard to adopt in single-author/single-user situations, which includes most of the intended audience for this paper.

One important observation about this list is that many experienced programmers actually do some or all of these things even for small projects. It makes sense for them to do so because (a) they've already paid the learning cost of the tool, so the time required to implement for the "next" project is small, and (b) they understand that their project will need some or all of these things as it scales, so they might as well put it in place now.

The problem comes when those experienced developers give advice to novices who *haven't* already mastered the tools, and *don't* realize (yet) that they will save time if and when their project grows. In that situation, advocating unit testing with coverage checking and continuous integration is as likely to scare novices off than to aid them.

[23] `http://dublincore.org/`
[24] `http://zotero.org/`
[25] `http://orcid.org/`

9 Conclusion

write a conclusion

References

1. Wilson G, Aruliah DA, Brown CT, Hong NPC, Davis M, et al. (2014) Best practices for scientific computing. PLoS Biology 12: e1001745.

2. Gentzkow M, Shapiro JM (2014). Code and data for the social sciences: A practitioner's guide. URL `http://faculty.chicagobooth.edu/matthew.gentzkow/research/CodeAndData.pdf`.

3. Noble WS (2009) A Quick Guide to Organizing Computational Biology Projects. PLoS Computational Biology 5.

4. Brown CT (2015). How to grow a sustainable software development process. URL `http://ivory.idyll.org/blog/2015-growing-sustainable-software-development-process.html`.

5. Wickham H (2014) Tidy data. Journal of Statistical Software 59: 1–23.

6. Kitzes J (2016). Reproducible workflows. URL `http://datasci.kitzes.com/lessons/python/-reproducible_workflow.html`.

7. Sandve GK, Nekrutenko A, Taylor J, Hovig E (2013) Ten simple rules for reproducible computational research. PLoS Computational Biology 9.

8. Hart E, Barmby P, LeBauer D, Michonneau F, Mount S, et al. (2015). Ten simple rules for digital data storage. doi:doi.org/10.7287/peerj.preprints.1448v1.

9. Orwell G (2002) Essays. Everyman's Library Classics.