Good Enough Practices for Scientific Computing

Greg Wilson[1,*], Jenny Bryan[2], Karen Cranston[3], Justin Kitzes[4], Lex Nederbragt[5], Tracy K. Teal[6]
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
6) Data Carpentry / tkteal@datacarpentry.org
∗ 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" [1]. It was well received, but 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 presents a set of "good enough" practices[3] for scientific computing, i.e., a minimum set of tools and techniques that we believe every researcher can and should adopt. It draws inspiration from many sources [2–8], from our personal experiences, and from the experiences of the thousands of people who have taken part in Software Carpentry 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 ready to move beyond saving spreadsheets with names like `results-updated-3-revised.xlsx` in Dropbox. A practice is included in our list if large numbers of researchers use it, and large numbers of people are *still* using it months after first trying it out. We include the second criterion because there is no point recommending something that people won't actually adopt.

Many of our recommendations are for the benefit of the collaborator every researcher cares about most: their future self[4]. If researchers don't see those benefits quickly enough to pay for the slowdown that inevitably occurs when adopting something new, they will almost certainly switch back to their old way of doing things. This rules out many practices such as code review that we feel are essential for larger-scale development (Section 8).

KC: Add pointers to papers or projects that follow these rules.

Acknowledgments

Our thanks to Arjun Raj (University of Pennsylvania) and Steven Haddock (Monterey Bay Aquarium Research Institute) for their feedback on early versions of this paper, and to everyone involved in Data Carpentry and Software Carpentry for everything they have taught us.

2 Data Management

Data within a project may need to exist in various forms, ranging from what first arrives to what is included in the final publication. The aim of our recommendations is to reproducibly create "tidy data", which can be a powerful accelerator for analysis [5, 8].

[1] `http://software-carpentry.org/`
[2] `http://datacarpentry.org/`
[3] Note that English lacks a good word for this: "mediocre", "adequate", and "sufficient" aren't exactly right.
[4] As the joke goes, yourself from three months ago doesn't answer email...

1. *Save data in the raw form.* Where possible, save data as originally generated (i.e. by an instrument or from a survey). 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; to recover from analytical mishaps; and for experimenting without fear. 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.

   If the data is large and in a format that is inefficient to store or process, transform it for storage with a lossless, well-documented procedure. Avoid devoting resources to storage of stable, long-lived repositories (i.e., don't save local copies of GenBank).

2. *Create the data you wish to see in the world.* Create a new version of the raw data that is the dataset you *wish* you had received. Use this dataset as the starting point for downstream analyses. This is the time to improve machine and human readability, but *not* to do vigorous data filtering or adding external information. Approach this step as non-destructive, both of the data and its general "shape".

   JB: Include an example.

   File formats: Save the data in open, non-proprietary formats that ensure long-term machine readability. CSV is good for tabular data, JSON, YAML, or XML for non-tabular data such as graphs[5], and HDF5 for certain kinds of structured data. This guide[6] has a useful guidelines for choosing file formats.

   TT: Include example of what goes wrong if you don't do tidying right.

   Variable names: Replace inscrutable variable names and artificial data codes with self-explaining alternatives, e.g., rename variables called `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 for missing data, such as "-99", with `NA`s, a shorthand code for "Not Available" that is used as a special variable in many programming languages [9].

   Filenames: Store 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 or by location.

3. *Create analysis-friendly data.* Re-formatting data will eliminate processing steps between loading the data set and doing analyses.

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

   Combine columns that only contain one variable's worth of information when taken together. 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, plus an additional column that indicates the time point.

   Reformat values to match your environment's built-in parsing rules. For example, use a date-time format that will be recognized automatically, or better yet, split dates into three columns (year, month, and day) for easier sorting and filtering.

[5] The node-and-arc kind.

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

OpenRefine

OpenRefine[7] 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. The cleanup process does not affect the original raw data, which further aids reproducibility.

4. *Record all the steps used to process data*: Data manipulation is as much a part of your analysis as any statistics. If you are not able to document this step clearly, it will not be possible for you, or anyone else, to repeat your analysis. As important, how will you know in a month whether you did it right the first time?

   Record all of the information needed to redo the processing of raw data. The best way to do this is to write scripts for *every* stage of data processing. This not only ensures that you have a record, but also allows you to re-do the cleanup and other steps when new data arrives. In cases where automated data processing is not possible or not practical, it's important to clearly document every manual action taken (what menu was used, what column was copied and pasted, what link was clicked, etc.). For example, choosing a region of interest in an image is inherently interactive, but you can save the region chosen as a set of boundary coordinates. If the exact details of the procedure cannot easily be recorded, save intermediate results for later checking.

5. *Give every record a unique key.* In a well-organized data set, it's easy to precisely refer to a specific record. 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 sorted.

   We therefore recommend giving each record a unique, persistent key that has no other purpose than identifying the record[8] The provision of such a key also makes it much easier to link data sets together.

6. *Include complementary data in additional files.* Raw data frequently does not fully explain itself. For example, if each well in an 8×12 microtitre plate is used to study a specific gene knockout, the raw data will have a `well` variable taking on values like "A1" and "G12". Create a supplementary file that relates the experimental conditions to each variable in the raw data. Use the same names and codes when variables in two datasets refer to the same thing to simplify merging and lookup, and to reduce errors.

7. *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). Sites such as Figshare[9], Dryad[10], and Zenodo[11] allow others to find your work, use it, and cite it; we discuss licensing in Section 4 below.

Taken in order, the recommendations above will produce intermediate data files with increasing levels of cleanliness and task-specificity. An alternative approach to data management would be to fold all data management tasks into the procedure, in particular the software, that is used for data analysis, so that intermediate data products are created "on the fly" and stored only in memory, not saved as distinct files.

[8] Student numbers and similar identifiers serve this purpose in institutional databases.
[9] `https://figshare.com/`
[10] `http://datadryad.org/`
[11] `https://zenodo.org/`

While the latter approach may be appropriate for projects in which very little data cleaning or processing is needed, we recommend that beginning computational researchers explicitly create and save these intermediate products for two reasons. First, saving intermediate files makes it easy to re-run *parts* of a data analysis pipeline, which in turn makes it easier and faster to tinker with specific data cleaning operations. Second, dividing data cleaning into several consecutive steps and separating it from data analysis helps to conceptually divide a lengthy workflow into smaller conceptual chunks, making it easier to understand, share, describe, and modify.

Choosing Metadata

[10] provides a useful classification for metadata, with implications for how to represent it. She notes it is important to distinguish metadata about the dataset as a whole (e.g., author and related publications) from metadata about the content (e.g., units for observations). When considering how to store metadata, consider the intended audience. If it is human beings, write a README file. If it is machines, such as metadata harvesters and formal repositories, create a metadata file that they will easily be able to parse.

3 Software

If you or your group are creating tens of thousands of lines of software for use by hundreds of people you have never met, you are doing software engineering. If you're writing a few dozen lines now and again, and are probably going to be its only user, you may not be doing engineering, but you can still make things easier on yourself by adopting a few key engineering practices. What's more, adopting these practices will make it easier for people to understand and (re)use your code.

The core realization in these practices is that *readable*, *reusable*, and *testable* are all side effects of writing *modular* code, i.e., of building programs out of short, single-purpose functions with clearly-defined inputs and outputs.

1. *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, a good example is worth a thousand words. Where possible, the comment should also indicate reasonable values for parameters. Figure 1 presents an example of of such a comment.

2. *Decompose programs into functions* that are no more than one page (about 60 lines) long, 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 items at once [11]. If we are to understand what our software is doing, we must break it into chunks that obey this limit, then create programs by combining these chunks.

3. *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.

   The easiest code to debug and maintain is code you didn't actually write, so *always search for well-maintained libraries that do what you need* before writing new code yourself, and *test libraries before relying on them.*

4. *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

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

Usage: generate_images.py -f fuzzing -n p_flaws -o output_file -r p_radius -s rand_seed -v -w 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)
```

Figure 1. A Good Comment To Explain The Use Of A Program

its name should be: while it's acceptable to call the counter variable in a loop `i` or `j`, the major data structures in a program should *not* have one-letter names.

> **Tab Completion**
>
> Almost all modern text editors provide *tab completion*, so that typing the first part of a variable name and then pressing the tab key inserts the completed name of the variable. Employing this means that meaningful longer variable names are no harder to type than terse abbreviations.

5. *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.

6. *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.

7. *Provide a simple example or test data set* that users (including yourself) can run to determine whether the program is working and whether it gives a known correct output for a simple known input. Such a "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.

8. *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. DOIs for software are provided by Figshare[12] and Zenodo[13].

4 Collaboration

You may start working on projects by yourself or with a small group of collaborators you already know, but you should design it to make it easy for new collaborators to join. These collaborators might be

[12] `https://figshare.com/`
[13] `https://zenodo.org/`

new grad students or postdocs in the lab, or they might be *you* returning to a project that has been idle for some time. As summarized in [12], you want to make it easy for people set up a local workspace so that they *can* contribute, help them find tasks so that they know *what* to contribute, and make the contribution process clear so that they know *how* to contribute. You also want want to make it easy for people to give you credit for your work.

1. *Create an overview of your project.* Have a short file in the project's home directory that explains the purpose of the project. This file (generally called `README`, `README.txt`, or something similar) should contain the project's title, a brief description, up-to-date contact information, and an example or two of how to run various cleaning or analysis tasks. It is often the first thing users of your project will look at, so make it explicit that you welcome contributors and point them to ways they can help.

   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, i.e., dependencies that need to be installed, tests that can be run to ensure that software has been installed correctly, and guidelines or checklists that your project adheres to.

2. *Create a shared public "to-do" list.* This can be a plain text file called something like `notes.txt` or `todo.txt`, or you can use 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 you choose, describe the items clearly so that they make sense to newcomers.

3. *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 Creative Commons licenses for data and text, either CC-0[14] (the "No Rights Reserved" license) or CC-BY[15] (the "Attribution" license, which sharing and reuse but requires people to give appropriate credit to the creators). For software, we recommend a permissive license such as the MIT, BSD, or Apache license [13].

> **What Not To Do**
>
> 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, she will be unable to include your data if the license says "non-commercial". We recommend permissive software licenses rather than the GNU General Public License[16] (GPL) because it is easier to integrate permissively-licensed software into other projects.

4. *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. Figure 2 shows the `CITATION` file for the Ecodata Retriever[17]; for an example of a more detailed `CITATION` file, see the one for the khmer[18] project.

[14] `https://creativecommons.org/about/cc0/`
[15] `https://creativecommons.org/licenses/by/4.0/`
[17] `https://github.com/weecology/retriever`
[18] `https://github.com/dib-lab/khmer/blob/master/CITATION`

```
Please cite this work as:

Morris, B.D. and E.P. White. 2013. "The EcoData Retriever:
improving access to existing ecological data." PLOS ONE 8:e65848.
http://doi.org/doi:10.1371/journal.pone.0065848
```

Figure 2. Example CITATION File

5 Project Organization

Organizing the files that make up a project in a logical and consistent directory structure will help you and others keep track of them. Our recommendations for doing this are drawn primarily from [2,3].

1. *Put each project in its own directory, which is named after the 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, while if you are building tools that are used in several projects, the common code should probably be in a project of its own. Anything in between can be decided based on the set of people you're collaborating with.

2. *Put text documents associated with the project in the* `doc` *directory.* This includes 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 in large projects.

3. *Put raw data and metadata in a* `data` *directory, and files generated during cleanup and analysis in a* `results` *directory*, where "generated files" includes intermediate results, such as cleaned data sets or simulated data, as well as final results such as figures and tables.

   The `results` directory will *usually* 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.

4. *Put project source code in the* `src` *directory.* `src` contains all of the code written for the project. This includes programs written in interpreted languages such as R or Python; those written compiled languages like Fortran, C++, or Java; as well as shell scripts, snippets of SQL used to pull information from databases; and other code needed to regenerate the results.

   This directory may contain two conceptually distinct types of files that should be distinguished either by clear file names or by additional subdirectories. The first type are files or groups of files that perform the core analysis of the research, such as data cleaning or statistical analyses. These files can be thought of as the "scientific guts" of the project.

   The second type of file in `src` 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".

5. *Put external scripts, or compiled programs in the `bin` directory.* `bin` contains scripts that are brought in from elsewhere, and executable programs compiled from code in the `src` directory[19]. Projects that have neither will not require `bin`.

> **Scripts vs. Programs**
>
> We use the term "script" to mean "something that is executed directly as-is", and "program" to mean "something that is explicitly compiled before being used". The distinction is more one of degree than kind—libraries written in Python are actually compiled to bytecode as they are loaded, for example—so one other way to think of it is "things that are edited directly" and "things that are not".

6. *Name all files to reflect their content or function.* For example, use names such as `bird_count_table.csv`, `manuscript.md`, or `sightings_analysis.py`. Do *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.

Figure 3 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, a `CITATION` file that explains how to reference it, and a `LICENSE` file that states the licensing. 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.

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 Keeping Track of Changes

Keeping track of changes that you or your collaborators make to data and software is a critical part of research. Being able to reference or retrieve a specific version of the entire project aids in reproducibility for you leading up to publication, when responding to reviewer comments, and when providing supporting information for reviewers, editors, and readers.

We believe that the best tools for tracking changes are the version control systems that are used in software development, such as Git, Mercurial, and Subversion. They keep track of what was changed in a file when and by whom, and synchronize changes to a central server so that many users can manage changes to the same set of files.

Although all of the authors use version control daily for all of their projects, we recognize that many newcomers to computational science find version control to be one of the more difficult practices to adopt. We therefore recommend that projects adopt *either* a systematic manual approach for managing changes *or* version control in its full glory.

[19]The name `bin` is an old Unix convention, and comes from the term "binary"

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

Figure 3. Example Project Layout

Whatever system you chose, we recommend that you use it in the following way:

1. *Back up (almost) everything created by a human being 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.

2. *Keep changes small.* Each change 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. 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.

3. *Share changes frequently.* Everyone working on the project should share and incorporate changes from others on a regular basis. 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 versioning procedure describe below, which does not provide any assistance for merging simultaneous changes.

4. *Create, maintain, and use a checklist for saving and sharing changes to the project.* The list should include writing log messages that clearly explain any changes, the size and content of individual changes, style guidelines for code, updating to-do lists, and bans on committing half-done work or broken code. See [14] for more on the proven value of checklists.

Manual Versioning

Our first suggested approach, in which everything is done by hand, has three parts:

1. *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 is repaid the moment a laptop is stolen or its hard drive fails.

2. *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). This file is the equivalent of a lab notebook, and should contain entries like those shown in Figure 4.

```
## 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).
```

Figure 4. Example Changelog

```
.
|-- 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
```

Figure 5. Directory Hierarchy for Manual Versioning

3. *Copy the entire project whenever a significant change has been made* (i.e., one that materially affects the results), 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 as shown in Figure 5. 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.

> **Data is Cheap, Time is Expensive**
>
> Copying 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 satisfies the 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.

Version Control Systems

What the manual process described above requires most is self-discipline. The version control tools that underpin our second approach—the one all authors 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.

How Version Control Systems Work

A version control system stores snapshots of a project's files in a repository. Users modify their working copy of the project, and then save 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. Tool-based version control has several benefits over manual version control:

- Instead of requiring users to make backup copies of the whole project, version control safely stores just enough information to allow old versions of files to be re-created on demand.
- Instead of relying on users to choose sensible names for backup copies, the version control system timestamps all saved changes automatically.
- Instead of requiring users to be disciplined about completing the changelog, 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.
- Instead of simply copying files to remote storage, version control checks to see whether doing that would overwrite anyone else's work. If so, they facilitate identifying conflict and merging changes.

It's hard to know what version control tool is most widely used in research today, but the one that's most talked about is undoubtedly Git[20]. This is largely because of the popularity of GitHub[21], a hosting site that provides free repositories to those willing to make their work openly accessible. For those who find Git's command-line syntax inconsistent and confusing, Mercurial[22] is a good choice; Bitbucket[23] provides free hosting for both Git and Mercurial repositories, but does not have nearly as many scientific users.

What Not to Put Under Version Control

Despite the benefits of version control systems, some types of files may *not* be appropriate for version control.

First, today's version control systems are not designed to handle megabyte-sized files, never mind gigabytes, so large data or results files should not be included. Raw data should not change, and therefore should not require version tracking. Keeping intermediate data files and other results under version control is not necessary if you can re-generate them from raw data and software. However, if data and results are small, we do recommend placing them under version control for ease of access by collaborators and for comparison across versions.

Second, some data formats are unfortunately not amenable to version control, which is designed to work with plain text files such as source code. In particular, Microsoft Office files (like the `.docx` files

[20] `https://git-scm.com/`
[21] `http://github.com`
[22] `https://www.mercurial-scm.org/`
[23] `https://bitbucket.org/`

used by Word or the `.xlsx` files used by Excel) can be stored in a version control system, but specific changes can't be tracked. 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.

Inadvertent Sharing

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.

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 writing is rarely addressed in discussions of scientific computing, computing has changed scientific writing just as much as it has changed research.

A common practice in academic writing 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. This results in a lot of files to keep track of, and a lot of tedious manual labor to merge comments to create the next master version.

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 are to:

- 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.
- Reduce the chances of work being lost or people overwriting each other's work.
- Make it easy to track and combine contributions from multiple collaborators.
- Avoid duplication and manual entry of information, particularly in constructing bibliographies, tables of contents, and lists.
- Make it easy to regenerate the final published form (e.g., a PDF) and to tell if it is up to date.
- Make it easy to share that final version with collaborators and to submit it to a journal.

The First Rule Is...

The workflow you choose is less important than having all authors agree on the workflow *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.

Single Master Online

Our first alternative has two parts:

1. *Write manuscripts using online tools with rich formatting, change tracking, and reference management*, such as Google Docs. With the document online, everyone's changes are in one place, and hence don't need to be merged manually.

2. *Include a `PUBLICATIONS` file in the project's `doc` directory* with 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 collaborators 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[24] can go a long way).

Text-based Documents Under Version Control

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

1. *Write the manuscript in a plain text format that permits version control* such as LaTeX[25] or Markdown[26], and then convert them to other formats such as PDF as needed using scriptable tools like Pandoc[27].

2. *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.

Using a version control system provides good support for finding and merging differences resulting from concurrent changes. It also provides a convenient platform for making comments and performing review.

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.

Why Two Recommendations for Manuscripts?

The first draft of this paper recommended always using plain text in version control 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.

[24] `http://pandoc.org/`
[25] `http://www.latex-project.org/`
[26] `http://daringfireball.net/projects/markdown/`
[27] `http://pandoc.org/`

Similarly, Arjun Raj said in a blog post[28]:

> 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.

In keeping with our goal of recommending "good enough" practices, we have therefore included online, collaborative editing 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.

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 We Left Out

We have deliberately left many good tools and practices off our list, including some that we use daily, because they only make sense on top of the core practices described above, or because it takes a larger investment before they start to pay off.

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[29] 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, newcomers 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

[28] `http://rajlaboratory.blogspot.ca/2016/03/from-over-reproducibility-to.html`
[29] `https://www.gnu.org/software/make/`

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[30] automatically run a set of user-defined commands whenever changes are made to a version control repository. These commands typically execute 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 changes are saved (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[31] are rarely encountered in the wild.

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[32], and should also obtain and use an ORCID[33] 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 [15].

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 people 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 more likely to scare newcomers off than to aid them.

[30] `https://travis-ci.org/`
[31] `http://dublincore.org/`
[32] `http://zotero.org/`
[33] `http://orcid.org/`

9 Conclusion

GW: 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. White EP, Baldridge E, Brym ZT, Locey KJ, McGlinn DJ, et al. (2013) Nine simple ways to make it easier to (re)use your data. Ideas in Ecology and Evolution 6.

10. Wickes E (2015). Comment on "metadata". URL `https://github.com/swcarpentry/good-enough-practices-in-scientific-computing/issues/3#issuecomment-157410442`.

11. Miller GA The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological Review 63: 8197.

12. Steinmacher I, Graciotto Silva M, Gerosa M, Redmiles DF (2015) A systematic literature review on the barriers faced by newcomers to open source software projects. Information and Software Technology 59.

13. St Laurent AM (2004) Understanding Open Source and Free Software Licensing. O'Reilly Media.

14. Gawande A (2011) The Checklist Manifesto: How to Get Things Right. Picador.

15. Petre M, Wilson G (2014) Code review for and by scientists. In: Katz D, editor, Proc. WSSSPE 2014.