Organization of a scientific computational project

Core guiding principle: Someone unfamiliar with your project (including you, some months later) should be able to look at your files, understand what you did and why, and easily repeat it.

Murphy's Law for computational projects: For anything you do, you will probably have to repeat it again later (new data, new parameters, fixing bugs).

General idea: Breaking a lengthy workflow into pieces makes it easier to understand, share, describe, and modify. Make your project modular, the modules being transparent and reproducible.

Fun fact: Applying good practices in a scientific computational project can save you A LOT OF TIME later and make you a much happier person.

Organization of files and directories

Stick to one standardized and self-explanatory directory structure; suggested organization:

- **`/Project_name`**
  - **`notes.txt`** (project notes/lab notebook, chronologically organized, possibly managed by a version control system; all project-related notes, e.g. information from collaborators, observations, conclusions, new ideas, links to other documents like images and tables; useful for meetings/progress reports; a simple text file is often good enough, but can be a more complex format)
  - additional possibly useful files: `requirements.txt` (defines computational environment, tools/programs/databases), `todo.txt` (to-do list), `README` (information explicitly intended for project users/collaborators)
  - **`/doc`** (project-related documents, e.g. slides, images, pdfs and manuscripts)
    - `/manuscript1`
  - **`/bin`** (or `/src`, project-related code/scripts; if you keep your scripts in a system-wide location, you might want to copy those that you use for the project)
  - `runall.sh` script: some people use a universal driver script that runs all other scripts and generates all results; this is useful, but can be less flexible than separate experiments with their own driver scripts as suggested below
  - **`/data`** (raw data and related metadata; make it read-only)
    - `/sample1` (logical organization) or e.g. `20180130` (chronological)
    - `/sample2` etc.
  - **`/01_experimentX`** (e.g. `01_filter`) or `/20190315_experimentX` (file name starting with numeric identifier or date)
    - **`01run.sh`** (main driver script, transforms files in `./data` into files in `./results`, completely automates *all* required steps; often combines calls to shell commands, self-written Python scripts and precompiled programs)
    - `02summarize.sh` (a script evaluating the results)
    - `/bin` (experimentX-specific scripts, may also reside in project-level /bin)
    - **`/data`** (input data for experimentX, may be symbolic links to project-level `/data/`… files or `/results/`… files from other experiments)
    - **`/results`** (results of experimentX)
    - **`notes.txt`** (notes regarding experimentX – motivation for this experiment, explanation what happens here, conclusions, ideas, etc., but also ***version number or date of download*** of used tools and databases)

        - `/temp` (temporary files, deleted upon completion of the experiment)
    - **`/02_experimentY`** (e.g. `02_parse`)
    - **`/03_experimentZ`** (e.g. `03_visualization`) etc.

Driver script

- **Automates every data processing step** like creating the required directory structure
- Records **every performed operation** (beware of manual editing of output files)
- Contains many **informative comments**
- Contains **all relevant information** like file and directory names and passes them as arguments to other scripts/programs
- Stores **paths and constants as variables in a separate section** in the beginning, which makes overview and modifications easier
- Uses **relative paths** (if project folder is transferred to another location, it still works)
- Checks **data consistency/plausibility** to make sure that things go as expected
- Uses `if (output_file does not exist), then <perform operation>` constructs to easily repeat parts of the experiment (after deletion of the respective files)
- The **environment** in which the driver script operates should be **clearly defined** (required tools and databases, and their versions)
- **Script should actually work** if run in the defined environment ☺

Alternative: Jupyter notebook

- **Jupyter notebooks** are a **useful data science tool** if used properly (should be well named, clearly structured, informatively commented), also for **sharing workflows and results**
- not suited for every task

Scripts in general

- Every script should provide **usage information, no matter how short it is** (can be a brief comment section in the beginning)
- Scripts should **break immediately if something is wrong:** check data consistency as often as possible (assertions in Python), e.g. input arguments, non-empty files, plausible results, …
- Adopt **iterative and incremental development**: start with a minimalistic working version of the script, keep changes small, test frequently
- Write modular code, i.e., **short, single-purpose functions/scripts** with clearly defined inputs and outputs -> readable, reusable, and testable; avoid “swiss-army-knife” scripts
- Provide **simple examples**/test data to make sure that the script works as expected
- **Avoid code duplication** (copying/pasting code is usually a bad sign)
- Look for **well-maintained libraries** that help you do what you’re trying to do
- Follow **best practices** like naming conventions (e.g. https://realpython.com/python-pep8/)

Code development and version control

- A version control system stores snapshots of a project’s files in a repository
- Provides **backup,** keeps track of **changes** (=versions with tags), allows **code modifications (branches)** without breaking existing functionality, facilitates conflict resolution
- If you **collaborate** with others, you need to **use version control**
- If you work on a **larger codebase**, you should **use version control**

Collaborating on computational projects

- Decide early on methods for communication/information exchange
- Use a version control system (Git is a good option) to manage changes to a project
    - Raw data and intermediate files need not be put under version control
    - Large data or result files should not be put under version control
- Keep changes small (= group of edits that you might want to undo in one step)
- Share changes frequently (synchronize your progress with the progress of others)
- Use an additional checklist (log file) for keeping track of and sharing changes to the project
- Store each project in a folder that is daily mirrored to Dropbox or a remote repository such as GitHub (and/or use some automated daily backup system)

Basic data management

- **Save raw data** in a separate directory (`/data`) and make it read-only
- Do not duplicate (data) files unless necessary, **use symbolic links** instead
- Keep **large files compressed** (gzip)
- Back up crucial files like raw data files, scripts and notes/documentation in at least **two spatially distinct locations** (an external drive next to the computer is a bad idea, e.g. in case of fire/theft/etc)
- **Make data analysis-friendly**: convert to open, non-proprietary, standardized formats that can be easily re-used later; modify cryptic variable names and file names to make them more informative; **tidy up the data** (http://garrettgman.github.io/tidying/)
- **Directory names**:
    - **chronological:** name is a date, e.g. `2018_03_15` (starts with year for better sorting), makes sense if you have many experiments of the same type differing by date
    - **logical:** name is an abstraction of the content, e.g. `assembly_firsttry`
    - **"semi-logical"** (often best option): name starts with number or date -> reflects a sequence of steps (e.g. `01_filter`, `02_parse`, `03_visualization`, etc.)
- **Name files to reflect their function or content** in chronological order, e.g. `dna_sample1.trimmed.filtered.assembled.filtered.fasta` (you can immediately see that the data was filtered before and after the assembly)
- **Temporary file names** should be distinct from permanent file names, so you know which files are incomplete or irrelevant (you can rename files in a subsequent step)
- **Delete temporary files**, especially if they can be easily recalculated (time/storage tradeoff)
- **Delete experimental files** after trying something out – keep your workspace clean; do it immediately after completion, it will be much harder later on
- **Archive inactive projects** in a separate directory, packed as .tar.gz archives
- **Share your data** using scientific online repositories

Basic project management

- Define **project goals**
- Outline the **milestones** necessary to reach the goals – each milestone should correspond to results (**deliverables**) that can be presented in form of a progress report/short presentation
- Outline the steps (**work packages**) required to reach each milestone
- Assess the **time and resources** required to complete each work package
    - Each work package should be disassembled into single steps, as fine-grained as possible (remember the algorithmic exercise about planning a city trip)

    - Think about possible difficulties and how you can overcome them (risk analysis)
- **Reserve time for work packages in a (online) calendar in advance**
- **Use the reserved time** to complete the work packages
    - introduce changes into the planning if required
- The **current project status** should be transparent to your PI and your collaborators

Basic manuscript management

- Agree with all authors on the workflow before the writing starts
- Keep a **single master document online** which allows to track changes and is available to all co-authors, using a platform such as **Google Docs**
- Alternatively, keep the manuscript in plain text format under version control, using LaTeX or Markdown
- Keep supplementary materials in separate text-format files for easier re-use by others

Beware of:

- **Manual modifications** of output files
    - Workflow becomes non-reproducible
    - You WILL forget later what you did
- **Messy workspace**
    - with lots of files of unclear importance lying around, taking up space and making you nervous (might sound funny, but this is exactly what happens)
- **Poorly tested/unjustified analysis steps**
    - Using non-default parameters, unpublished tools or untested workflows without very good reason -> this will be much, much harder to publish later
- "**Overfitting**"
    - Fine-tuning the workflow (e.g. tool parameters) to achieve the "desired" results
    - The results may become slightly better, but less reproducible and harder to publish
- **Confirmation bias** (human tendency to handle information in a way that confirms one's preexisting beliefs or hypotheses)
    - Don't "adapt" the analysis to your ideas about what the results should be; this might give seemingly better results short-term, but will cause more problems long-term
    - This is not the same as optimizing the workflow by identifying the tools/approaches best-suited for your data to obtain good results ☺
    - Rule of thumb: A good result is often stable towards changes in parameters and even analysis methods. If it is not, it might not be a reliable result.

Links

- **Software/Data Carpentry**: https://software-carpentry.org/, https://datacarpentry.org/
- Noble, William Stafford. 2009. "A Quick Guide to Organizing Computational Biology Projects." *PLOS Computational Biology* 5 (7): e1000424. https://doi.org/10.1371/journal.pcbi.1000424.
- Wilson, Greg et al. 2017. "Good Enough Practices in Scientific Computing." *PLOS Computational Biology* 13 (6): e1005510. https://doi.org/10.1371/journal.pcbi.1005510.