April 15, 2019
This is a preprint that has not undergone peer review yet.

Design guidelines for data analysis scripts

Marijn van Vliet[1]

[1]Department of Neuroscience and Biomedical Engineering, Aalto University, Finland, marijn.vanvliet@aalto.fi

Abstract

Unorganized heaps of analysis code are a growing liability as data analysis pipelines are getting longer and more complicated. This is worrying, as neuroscience papers are getting retracted due to programmer error. In this paper, some guidelines are presented that help keep analysis code well organized, easy to understand and convenient to work with:

1. Each analysis step is one script
2. A script either processes a single recording, or aggregates across recordings, never both
3. One master script to run the entire analysis
4. Save all intermediate results
5. Visualize all intermediate results
6. Each parameter and filename is defined only once
7. Distinguish files that are part of the official pipeline from other scripts

In addition to discussing the reasoning behind each guideline, an example analysis pipeline is presented as a case study to see how each guideline translates into code.

Keywords: data analysis, scripting, guidelines, programming

1 Introduction

The journey of the data from our measurement equipment to a figure in a publication is growing longer and more complicated. new preprocessing steps have been developed to be added at the beginning of the pipeline,[1] new multivariate techniques that find a place in the middle,[2] and new statistical methods at the end.[3] Using these new techniques often requires writing pieces of programming code referred to as "scripts", and in accordance with the growing data analysis pipelines, these scripts also tend to increase in length and complexity. When programming code becomes sufficiently convoluted, even the most experienced programmers will make mistakes, which can ultimately lead to erroneous conclusions. It has happened that papers had to be retracted due to programmer error.[4] This paper is an effort to set some guidelines to manage the complexity of scripts.

In science, data analysis is performed at the cutting edge, where it is often inevitable that new pieces of programming code need to be written. New methods are made available through software libraries first, being accessible through an application programming interface (API), and only later as graphical user interface (GUI) programs, if at all. Furthermore, novel research ideas often require combining analysis techniques in new ways that are not possible with existing programs and hence require writing new code. When data analysis pipelines grow, unorganized heaps of code become a liability.

[1] Bigdely-Shamlo, Mullen, Kothe, Su, and Robbins, 2015; Nolan, Whelan, and Reilly, 2010; Jas, Engemann, Bekhti, Raimondo, and Gramfort, 2017

[2] Kriegeskorte, Mur, and a. Bandettini, 2008; King and Dehaene, 2014; McIntosh and Mišić, 2013

[3] Maris and Oostenveld, 2007

[4] Casadevall, Steen, and Fang, 2014; Miller, 2006; Merali, 2010

An important aspect to organizing scientific code is to have the bulk of the analysis functionality implemented in the form of a software library that exposes a well designed API.[5] The code can then be used in multiple scripts and user facing programs. It is generally beneficial to use and extend an existing piece of software that is used by many, rather than developing a home-grown solution from scratch. This is because the more people use a piece of software, the more likely it is that mistakes are spotted and corrected.[6] There is a large body of literature on managing complexity and reducing the chance for programmer error in this context.[7]

[5] Buitinck et al., 2013

[6] Eklund, Nichols, and Knutsson, 2016

[7] Beck, 2002; Hunt and Thomas, 1999; Martin, 2008; McConnell, 2004; Wilson et al., 2014

The guidelines in this paper aim to translate some of this literature to the sub-domain of analysis scripts. Scripts are pieces of code that use the functionality that is exposed by the APIs of software libraries to create data analysis pipelines according to the specific requirements of a single study. Scripts are pieces of code that do not need to be reusable (pieces that need to be reusable are better implemented in a software library), only have to function correctly on one specific dataset (hence there are no "edge cases") and will generally only be used by yourself and your collaborators (save the occasional run for review purposes and replication studies). Therefore, many of the standard practices of the software industry do not apply or need translation in order to arrive at concrete advice of what to do and what to avoid when writing analysis scripts.

Often, an analysis pipeline starts off as a simple script that runs a few operations and grows as more steps are added. As the pipeline becomes more complicated, the overall organization and design of the pipeline must be occasionally re-evaluated, or it is likely to become convoluted and error prone. The guidelines in this paper aim to facilitate a successful organization of the analysis code, thereby keeping the complexity of data analysis scripts within tolerable limits, capitalize on the advantages of scripting and offset the disadvantages.

To move beyond mere truisms, the analysis pipeline developed by van Vliet, Liljeström, Aro, Salmelin, and Kujala (2018) has been extended to implement all guidelines in this paper, and will be used as case study. In the case study, the practical consequences are discussed of each guideline in terms of code, which can serve as an example when implementing your own analysis pipelines.

The example pipeline starts from the raw magnetoencephalography (MEG) and structural functional magnetic resonance imaging (fMRI) data from the Wakeman and Henson (2015) faces dataset and performs several artifact reduction steps, source estimation, functional connectivity analysis, cluster permutation statistics and various visualizations. The size and complexity of the pipeline is representative of that of the pipelines in modern studies at the time of writing. Where van Vliet et al. (2018) gives a detailed explanation of all analysis steps, the current paper focuses on the design decisions that were made during the implementation. You can find the code repository for the analysis pipeline at: https://github.com/aaltoimaginglanguage/conpy. Of special interest is the `scripts` folder of that repository, which contains the analysis code itself.

The "Application of the guideline to the example analysis" sections refer frequently to the example code and it is recommended to study these sections and the code side by side. The electronic version of this document contains many hyperlinks to sections of the code, which the reader is encouraged to follow to see how the guidelines can be implemented in practice. Hyperlinks are typeset in dark blue.

2 Guidelines

2.1 Guideline 1: Each analysis step is one script

An effective strategy to reduce software complexity is to break up a large system into smaller parts.[8] The first guideline is therefore to isolate each single step of an analysis pipeline into its own self-contained script. This greatly reduces complexity by allowing us to reason about the pipeline on two levels. At the lower level, we can reason about the implementation of a single step, while ignoring the rest of the pipeline for a moment. At the higher level, we can treat the individual steps as "black boxes" and focus on how they are combined together to form the complete pipeline (see also guideline 3), ignoring their implementation for a moment.

[8] Hofmann, 2004; Parnas, 1972

Things that this guideline aims to prevent:

- script becomes "spaghetti code"
- excessive running time of the script
- parts of the script are commented out in order to skip a time-consuming step
- if-statements being used to toggle parts of the script on and off

This raises the question of what exactly constitutes a single analysis step. The decision of where to "cut" the pipeline can be made from different perspectives: by complexity, by theme, and by running time.

Complexity perspective

The purpose of the guideline is that each individual script should be easy to understand and reason about, so one way to define a single step is by its complexity: if a single script becomes too complex to be easily understood as a whole, it should be split up into smaller steps if possible.

Thematic perspective

Ideally, understanding one script should not require knowledge of another script. If each script can be viewed as a self-contained box that performs a single task, the pipeline as a whole becomes simply a collection of these boxes that are executed in a specific order. A script should therefore aim to implement a single task, not multiple, and implement it completely, not only part of it. However, this aim may clash with other perspectives in this list, so compromises are sometimes necessary.

Time perspective

While running the entire analysis pipeline may take days, a single script should finish in a reasonable time. This invites frequent testing as you iteratively develop the script, allows you to quickly evaluate the effect of a parameter, and also makes it painless to ensure that the latest version of the script matches the latest result. When the running time grows to the point where you are tempted to continue working on the script while a run is still in progress, the script should be split into smaller steps if possible.

Application of the guideline to the example analysis

The van Vliet et al. (2018) pipeline consists of 13 analysis scripts that process the data, 4 visualization scripts that construct the figures used in the publication, a configuration file (`config.py`) and a "master" script that calls the individual analysis scripts (`dodo.py`, see guideline 3). Each of the 13 analysis scripts implements a single step in the analysis and are numbered to indicate the sequence in which they are designed to be run. While the scripts need to be run in sequence once, they can be run independently afterwards. The analysis scripts are relatively short (Figure 1), containing an average of 40.8 lines of code (std. 14.8), while the configuration and master scripts are longer.

Often, the reasoning behind the scope of each script was made from a thematic perspective.

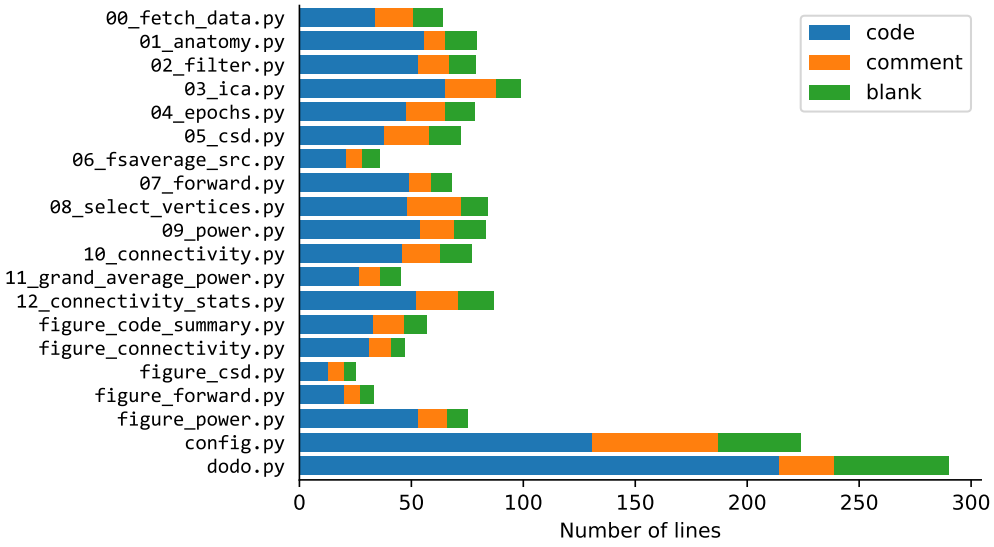

Figure 1: For each script in the analysis pipeline, the number of lines of the file, broken down into lines of programming code (code), lines of descriptive comments (comment) and blank lines (blank). The first 13 scripts perform data analysis steps, the next 4 scripts generate figures, the `config.py` script contains all configuration parameters and the `dodo.py` script is the master script that runs all analysis steps on all recordings.

For example, one script performs the source estimation (`09_power.py`) and another the connectivity estimation (`10_connectivity.py`). However, the decision to split the artifact reduction steps into two scripts (`02_filter.py`, `03_ica.py`) was made from a time perspective. Since the independant component analysis (ICA) computation takes time, it was split off into its own script to avoid having to repeat it unnecessarily. Finally, the decision to split up the construction of the forward models (i.e. leadfield) into three steps (`06_fsaverage_src.py`, `07_forward.py`, `08_select_vertices.py`) was made from a complexity perspective.

2.2 Guideline 2: A script either processes a single recording or aggregates across recordings, never both

A compelling reason for performing data analysis using scripts instead of, for example, using a GUI, is the ease of repeating (parts of) the analysis. Every time the scripts are run, the computer will perform exactly the same tasks in exactly the same order, eliminating the possibility of mix-ups in this regard. This allows you, for example, to efficiently test the effect of changing a single parameter, while ensuring all subsequent analysis steps remain the same. To capitalize on this advantage, analysis scripts should be organized such that it is easy to run only selected parts of it, without having to modify (e.g. "commenting out") the code itself.

Things that this guideline aims to prevent:

- excessive running time of the script
- multiple versions of the Big Loop that operate on different sets of subjects, all but one commented out
- a copy/paste of the Big Loop for each analysis script

In neuroscience, it is common to apply the same data processing steps to multiple recordings. For example, a frequently seen construct is the Big Loop over data from multiple participants. The second guideline states a separation of duties: a script is either a processing script, or an aggregation script. Processing scripts perform data analysis only on a single recording, passed as a parameter from the command line, and do not have the Big Loop (the script is applied to all recordings in a separate "master" script (see guideline 3)). Aggregation scripts have the Big Loop to collect the processed data from multiple recordings, with the sole purpose of aligning the data (e.g. morphing to a template brain) and computing an aggregate (e.g. a grand average or statistics).

This reduces the complexity of the code, since it allows the reader to either focus on the

intricacies of a data processing step, without having to worry about how the data is later reconciled across recordings, or to focus solely on the details of how multiple datasets are aligned and combined.

Following the guideline also makes the development process more efficient. It allows for a smooth workflow for the common scenario in which the script is tested on one subject during development, then an attempt is made to run it on all subjects using the master script, problems are found that only arise for certain subjects, and finally the script is re-run once more on all subjects.

Application of the guideline to the example analysis

In the example pipeline, there is a strict separation between scripts that perform data analysis on a single participant (steps 0–10) and scripts that aggregate across participants (steps 11 and 12). The scripts implementing steps 0–10 all take a single command line parameter indicating the participant to process. This is implemented with the `argparse` module of the standard Python library, which facilitates the generation of a helpful error message when this parameter is omitted, along with documentation on how to run the script. Not only does this help to keep the number of lines of code and the running time of the script down (Figure 1), it also opens up the possibility for the master script (see guideline 3) to automatically skip running the script for participants that have already been processed earlier.

2.3 Guideline 3: One master script to run the entire analysis

Once the individual steps have been implemented as a collection of scripts, the pipeline can be assembled in a "master" script that runs all the steps on all the data. This master script is the entry point for running the entire analysis and therefore the third guideline states that there should ideally only be one such script.

By having a strict separation between the scripts that implement the individual steps and the master script, it becomes possible to view the pipeline on two levels: the implementation details of each single step, and how the steps fit together to build the pipeline. To understand the latter, the master script provides the "floor plan" of the analysis, which can be studied without having to go into detail on how each individual step is performed. Hence, the only function that the master script should perform is to call the other scripts in the correct order. Actual data analysis steps, including logic for combining results across scripts, should always be performed in a separate script, which is in turn called from the master script.

Speed is a very important aspect of an analysis pipeline, as it encourages practices that reduce the likelihood of errors. Speed encourages incremental development, running the pipeline often during development to check the intermediate results. Speed encourages exploration, trying different parameters and approaches to obtain the best result possible. And last but not least, speed encourages re-running a script every time it has changed.

Apart from using efficient algorithms, the key to obtaining speed is to never repeat a time consuming calculation unnecessarily. If all analysis steps are properly isolated from one another (see guidelines 1 and 2) and all intermediate results are properly stored (see guideline 3), analysis steps for which the corresponding scripts have not changed, need not be run again if a script further down the pipeline has changed.

Things that this guideline aims to prevent:

- excessive complexity
- confusion as to which order the scripts should be run in
- having to manually run several scripts in order to complete the analysis and forgetting to re-run one
- scripts are changed, but not re-run, causing the result to be out of sync with the code
- excessive running time of the analysis pipeline when only a single step has changed
- a copy/paste of the Big Loop for each analysis script

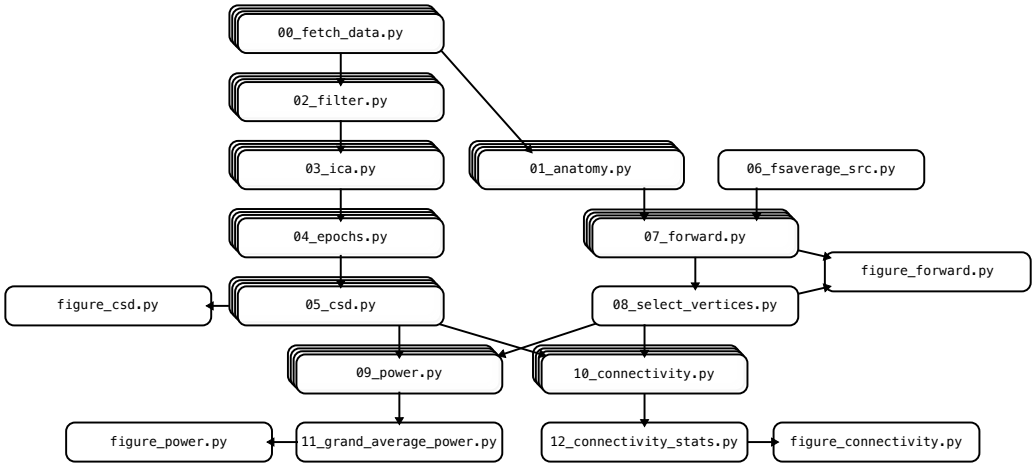

Figure 2: Dependency graph showing how the output of one script is consumed by another. Stacked boxes indicate scripts that are run for each participant.

A compelling advantage of scripting is that the code serves as a complete transcript of exactly what analysis steps were performed. However, this transcript is only correct if the latest version of the code is also the version that was used to produce the latest results. During the development of the scripts, we commonly make changes, re-run the script, inspect the result, and make more changes. If we are not careful, the code and results may become desynchronized, especially when multiple versions of the code and results are in play simultaneously. Putting misplaced trust in a wrong transcript can be very frustrating when attempting to reproduce a result.

Keeping track of which scripts have changed since they were last ran, and which scripts consume the output produced by which other scripts (known as the dependency graph, see figure Figure 2), is a task that has been studied in great detail in the area of software engineering and many specialized tools, known as "build systems",[9] are available to perform the required bookkeeping tasks. Writing the master script using a build system will allow fine grained control over which steps to run on which recordings, while skipping steps that are "up to date".

Application of the guideline to the example analysis

The master script of the example pipeline, `dodo.py`, is implemented using the `pydoit` [10] build system. In the script, all analysis steps are described as "tasks", which steps 0–10 having a "subtask" for each participant. Each task is associated with one of the analysis scripts, along with a list of files the script consumes and produces. This allows the build system to work out the dependency graph of the analysis pipeline (Figure 2).

The build system keeps track of which tasks are "up to date", meaning the latest version of the analysis scripts of all analysis steps up to and including the current step have all been run. This means that the entire analysis pipeline can be run often and cheaply: all steps that are up to date will be skipped. Making a change anywhere within the analysis code will prompt the recomputation of all the steps that need to be re-run.

The build system also provides a set of commands that allow for executing specific parts of the pipeline, for example, a single analysis step on all participants, a few specific steps on a few specific participants, etc., without having to change (e.g. comment out) any code.

[9] For example, here are some build systems that are optimized for creating data analysis pipelines:
https://snakemake.readthedocs.io
https://pydoit.org
https://luigi.readthedocs.io

[10] https://pydoit.org

2.4 Guideline 4: Save all intermediate results

First, from the complexity perspective, it is important that each script can function in isolation and does not rely on data that was left in memory by another script. The more each script can be isolated from the rest of the pipeline, the easier it is to understand and represent as a self contained black box.

The fourth guideline states that all intermediate results generated during the execution of a script should be saved to disk, if feasible. Having snapshots of the data as it passes through the pipeline has numerous advantages.

Another big advantage is that it makes it possible to skip any data processing steps that are unchanged since the last time the pipeline was executed (see also guideline 3). This makes it possible to re-run small portions of the pipeline quickly, for example to debug a problem, or to assess the effect of some parameter.

Finally, having all intermediate results readily available facilitates manual checks and exploration of the data. The ability to jump into an interactive session and quickly load the state of the data at any desired location in the analysis pipeline is an effective way to verify that a script produced the intended result.

Things that this guideline aims to prevent:

- variables being manipulated across multiple scripts
- debugging a script taking a long time due to having to re-compute everything from scratch every time the script runs
- erroneous output is generated in the middle of the script, but subsequent processing makes the result appear reasonable at the end of the script

Application of the guideline to the example analysis

In the example analysis pipeline, each script begins by loading the data that were produced by previous analysis steps as requires. Each script ends by saving all data that was produced by the script. This includes the processed MEG data, but also, for example, the ICA decomposition matrix, along with the indices of the ICA components that were judged to correspond to eye-blink contaminants.

2.5 Guideline 5: Visualize all intermediate results

Data analysis pipelines, such as those used in neuroscience, are sufficiently complex that failures should be expected and planned for. When designing the pipeline, think about the system for catching errors when they happen.

The most severe disadvantage of using scripts instead of a GUI may be the lack of direct feedback. Since the result is usually not immediately visualized, errors may stay hidden for a long time. Programming a computer is not unlike receiving a wish from a mischievous genie: you will get exactly what you asked for, but not necessarily what you wanted. As long as the final result of a series of processing steps looks reasonable, intermediate steps might contain nonsensical results that we would never know about unless we take care to check everything. Therefore, an analysis pipeline should invite frequent visual checks on all intermediate results.

The fifth guideline states that for each intermediate result, the script should create a visualization of the result and save it to disk. This does not need to be a publication ready figure, but must provide a visual confirmation that the data analysis operation had the intended result. By re-creating the figures every time a script is run and overwriting the file on disk, the figure remains up to date.

After running all the analysis scripts, a complete visual record should be available of the data as

Things that this guideline aims to prevent:

- researcher is operating “in the blind”
- erroneous output is generated in the middle of the script, but subsequent processing makes the result appear reasonable at the end of the script
- result figures no longer match the data files after a script has been re-run

it moves through the pipeline. Care should be taken that the order of the figures matches that of the analysis steps. Such a record invites frequent visual checks of the obtained results and therefore somewhat offsets the main advantage that GUI programs have over scripting.

Application of the guideline to the example analysis

Whenever an intermediate result is saved to disk in the example analysis pipeline, a simple visualization is also created and added to a "report" file. The main analysis package used in the example pipeline, MNE-Python,[11] provides a `Report` class that compiles a set of figures into a single HTML file. Each script adds (and overwrites) figures to the same report, which will grow in length as more scripts are run. The resulting HTML file contains an easy to navigate visual record of the data flowing through the pipeline. Each participant has their own report file.

[11] Gramfort et al., 2013

2.6 Guideline 6: Each parameter and filename is defined only once

It is not uncommon that a parameter is used in multiple scripts. The sixth guideline states that the value of each parameter should be defined in one place. Instead of copying the value of a parameter into all scripts that need it, the parameter should be imported, i.e., the programmer specifies the location where the parameter is defined and the programming language will take care of fetching the value when it is needed. In the programming literature, this is referred to as the "don't repeat yourself" (DRY) principle.[12]

[12] Martin, 2008

Importing, rather than copying, eliminates a common source of errors. When we change the value of a parameter, we may not be aware that we need to change it in multiple locations (either we forgot about the copies or we didn't know about them in the first place), resulting in different values being used at different locations and hence errors that can be very difficult to spot as long as the final result looks reasonable.

Things that this guideline aims to prevent:

- the same parameter, defined at two locations, with two conflicting values
- when changing a parameter, not knowing where else in the code the same change should be made
- wasting time copy/pasting things across multiple scripts

A good tactic for managing parameters is to create a single configuration script, which sole function is to define the values for all parameters. It makes it obvious were to look for the definition of a parameter and decreases the chances of accidentally defining the same parameter at two different locations.

Filenames are also parameters, and ones that are commonly shared across scripts too: one script producing a file that another script consumes. Just like other parameters, the guideline mandates that all filenames should be defined once and imported (not copied) by scripts that need it. It is not uncommon for a filename to change when parts of the analysis pipeline are added or removed. By ensuring the change needs to be made in only one location, we can ensure that scripts are not consuming outdated files.

A good tactic for managing filenames is to define templates for them in the configuration file. The templates can have placeholders for things like the participant number or experimental condition. See the implementation example for a more thorough explanation.

Application of the guideline to the example analysis

The example pipeline has a central configuration script `config.py` which defines all relevant parameters for the analysis, such as filter settings, the list of subjects, the experimental conditions, and so forth. All analysis scripts import the configuration file and thereby gain access to the parameters. Whenever a parameter needs to be changed or added, the configuration file is the

single authoritative location where the edit needs to be made and the change is propagated to all analysis scripts.

The `config.py` script starts by offering some machine specific parameters, such as the number of CPU-cores to dedicate to the analysis and where on the disk the data is to be stored. The configuration script queries the hostname, so that different parameters can be specified for different machines.

Since the pipeline stores all intermediate results and their visualizations, there is a large number of filenames to deal with. In many cases, each filename is used four times: once in the script generating the file, once in the script consuming the file, and twice in the master script `dodo.py`. For this reason, a helper class `Filenames` has been written that offers an efficient way to manage them. The class is used to create an `fname` namespace that contains short aliases for all filenames used throughout the pipeline. It also leverages Python's native string formatting language to allow quick generation of lists of filenames that adhere to a pattern (e.g., `"sub01_raw"`, `"sub02_raw"`, `...`).

2.7 Guideline 7: Distinguish files that are part of the official pipeline from other scripts

The development of a complex analysis pipeline is seldom a straightforward path from start to finish. Rather, ideas get tried and discarded, mistakes are made and fixed, and smaller analysis are made on the side. This all causes a tendency for scattering miscellaneous files around, littering the folders that contain the main pipeline, obfuscating what is relevant and what is not. However, since creative freedom is important, it would be counterproductive to strive for a perfectly clean workspace all the time. Rather, the occasional mess should be embraced, as well as the resulting responsibility to clean up after ourselves.

Things that this guideline aims to prevent:

- inability to distinguish incomplete or flawed scripts from proper ones
- not knowing what files are relevant to the main analysis
- not knowing what script is the master script that will run the entire analysis
- not knowing which version of the script was last run on the data

The seventh guideline calls for an organization system that distinguishes between files that are in a stable state and part of the main pipeline, and files that are work in progress, temporary, or part of analyses on the side. A good system reduces the effort of the cleaning process, making it easier to commit to a regular tidying up of the virtual workplace. It is important that the system does not become burdensome, as a simple system that is actually used is better than a more powerful one that is not.

This can be implemented in whatever way suits your workflow best, ranging from simply maintaining a rigid naming convention and folder structure, to using more powerful tools such as a version control system (VCS). Note that a VCS is not an organization system in itself, but merely a tool for implementing one. It is up to the data analyst to devise their own system and have the discipline to stick to it.

Application of the guideline to the example analysis

During the development of the pipeline, many scripts were written to try out different analysis approaches, conduct tests and do miscellaneous other tasks. The naming system was such that all analysis scripts that are officially part of the pipeline are either prefixed with a number (`00_`–`12_`) and all scripts that produce figures for the manuscript with `figure_`. From time to time, any script lacking such a prefix would be closely scrutinized to determine whether is was still relevant, and if not, deleted.

Deleted files were never truly gone though, as the project is managed by the VCS "Git".[13] Git

[13] https://git-scm.com

keeps track of the history of a file, allowing to return to previous versions, as well as parallel copies when doing something experimental. Although VCSs are primarily used to facilitate collaboration on a software project, they are useful even when working alone.[14] For one, they provide a crucial backup service, allowing recovery from mistakes and thus freedom to experiment and making bold decisions. Secondly, although VCSs do not impose any organizational structure for managing multiple copies of files, they facilitate creating and maintaining one.

[14] Vuorre and Curley, 2018

3 Conclusion

Following the given guidelines improves the chances of the analysis code being correct, by aiming for code that is easy to understand. The key to reducing the complexity of data analysis scripts is to cut up the pipeline into bite-sized chunks. Therefore, the guidelines state that each step of the pipeline should be implemented as a separate script that finishes in a reasonable time, has little to no dependencies on other scripts, writes all intermediate results to disk and visualizes them. In addition, one master script should exist that calls the other scripts in the correct order to execute the complete analysis pipeline.

Writing understandable code is a skill that can be honed by forming good habits. Whenever a problem arises in a pipeline, there is an opportunity to look beyond the specific problem to the circumstances that allowed to the problem to occur in the first place and the formation of new habits to prevent such circumstances in the future. However, it is important not to become bogged down in rules. Every new project is a chance for reviewing your habits: keep things that were beneficial and drop things that were not or which costs exceed their utility.

Be aware that there is a lot of software tooling available to automate repetitive tasks and perform bookkeeping. Whenever a rule needs enforcing or a repetitive action needs performing, there is likely a software tool available to automate it. However, while they can make it easier to adopt good habits and keep the code organized, they cannot do the job by themselves. Ultimately, it is up to the data analyst to keep things tidy and re-evaluate the design of the analysis pipeline as it grows. When and how to do this is best learned through experience. By reflecting at the end of each project what were good and bad design choices, the analysis pipeline of the next project will be better than the last.

4 Acknowledgements

Thanks goes out to all the members of the department of Neuroscience and Biomedical Engineering (NBE) at Aalto University who contributed to the discussion during the lab meeting concerning data analysis pipelines. MvV is supported by the Academy of Finland (grant 310988).

References

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley.

Bigdely-Shamlo, N., Mullen, T., Kothe, C., Su, K.-M., & Robbins, K. A. (2015). The PREP pipeline: standardized preprocessing for large-scale EEG analysis. *Frontiers in Neuroinformatics*, *9*(June), 16. doi:10.3389/fninf.2015.00016

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., … Varoquaux, G. (2013). API design for machine learning software: experiences from the scikit-learn project. *arXiv:1309.0239 [cs]*.

Casadevall, A., Steen, R. G., & Fang, F. C. (2014). Sources of error in the retracted scientific literature. *The FASEB Journal*, *28*(9), 3847–3855. doi:10.1096/fj.14-256735

Eklund, A., Nichols, T. E., & Knutsson, H. (2016). Cluster failure: why fMRI inferences for spatial extent have inflated false-positive rates. *Proceedings of the National Academy of Sciences*, *113*(28), 7900–7905. doi:10.1073/pnas.1602413113

Gramfort, A., Luessi, M., Larson, E., Engemann, D. A., Strohmeier, D., Brodbeck, C., … Hämäläinen, M. S. (2013). MEG and EEG data analysis with MNE-Python. *Frontiers in Neuroscience*, *7*(December), 1–13. doi:10.3389/fnins.2013.00267

Hofmann, M. A. (2004). Criteria for Decomposing Systems Into Components in Modeling and Simulation: Lessons Learned with Military Simulations. *SIMULATION*, *80*(7-8), 357–365. doi:10.1177/0037549704049876

Hunt, A. & Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Boston: Addison-Wesley Professional.

Jas, M., Engemann, D. A., Bekhti, Y., Raimondo, F., & Gramfort, A. (2017). Autoreject: automated artifact rejection for MEG and EEG data. *NeuroImage*, *159*, 417–429. doi:10.1016/j.neuroimage.2017.06.030

King, J.-R. & Dehaene, S. (2014). Characterizing the dynamics of mental representations: the temporal generalization method. *Trends in Cognitive Sciences*, *18*(4), 203–210. doi:10.1016/j.tics.2014.01.002

Kriegeskorte, N., Mur, M., & a. Bandettini, P. (2008). Representational similarity analysis - connecting the branches of systems neuroscience. *Frontiers in systems neuroscience*, *2*(November), 4. doi:10.3389/neuro.06.004.2008

Maris, E. & Oostenveld, R. (2007). Nonparametric statistical testing of EEG- and MEG-data. *Journal of Neuroscience Methods*, *164*(1), 177–190. doi:10.1016/j.jneumeth.2007.03.024

Martin, R. C. (2008). *Clean Code: a handbook of agile software craftmanship*. Prentice Hall.

McConnell, S. (2004). *Code Complete: a practical handbook of software construction* (2nd). Microsoft Press.

McIntosh, A. R. & Mišić, B. (2013). Multivariate statistical analyses for neuroimaging data. *Annual Review of Psychology*, *64*(1), 499–525. doi:10.1146/annurev-psych-113011-143804

Merali, Z. (2010). Computational science: ...Error...why scientific programming does not compute. *Nature*, *467*(7317), 775–777. doi:10.1038/467775a

Miller, G. (2006). Scientific publishing. A scientist's nightmare: software problem leads to five retractions. *Science (New York, N.Y.) 314*(5807), 1856–1857. doi:10.1126/science.314.5807.1856

Nolan, H., Whelan, R., & Reilly, R. B. (2010). FASTER: Fully Automated Statistical Thresholding for EEG artifact Rejection. *Journal of Neuroscience Methods*, *192*(1), 152–162. doi:10.1016/j.jneumeth.2010.07.015

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058. doi:10.1145/361598.361623

van Vliet, M., Liljeström, M., Aro, S., Salmelin, R., & Kujala, J. (2018). Analysis of Functional Connectivity and Oscillatory Power Using DICS: From Raw MEG Data to Group-Level Statistics in Python. *Frontiers in Neuroscience, 12*. doi:10.3389/fnins.2018.00586

Vuorre, M. & Curley, J. P. (2018). Curating research assets: a tutorial on the Git version control system. *Advances in Methods and Practices in Psychological Science, 1*(2), 219–236. doi:10.1177/2515245918754826

Wakeman, D. G. & Henson, R. N. (2015). A multi-subject, multi-modal human neuroimaging dataset. *Scientific Data, 2*, 150001. doi:10.1038/sdata.2015.1

Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., … Wilson, P. (2014). Best Practices for Scientific Computing. *PLOS Biology, 12*(1), E1001745. doi:10.1371/journal.pbio.1001745