Porcupine: a visual pipeline tool for neuroimaging analysis

Tim van Mourik[a,*], Lukas Snoek[b], Tomas Knapen[c,d], David G Norris[a,e]

[a]*Radboud University Nijmegen, Donders Institute for Brain, Cognition and Behaviour, Kapittelweg 29, 6525EN Nijmegen, The Netherlands*
[b]*University of Amsterdam, department of Brain & Cognition, Nieuwe Achtergracht 129B, 1018WS Amsterdam, The Netherlands*
[c]*Cognitive Psychology & Institute for Brain & Behavior, Vd Boechorststraat 1, 1081BT Amsterdam*
[d]*Spinoza Centre for Neuroimaging, Meibergdreef 75, 1105 BK Amsterdam*
[e]*Erwin L. Hahn Institute for Magnetic Resonance Imaging, Arendahls Wiese 199, 45141 Essen, Germany*

Abstract

The field of neuroimaging is rapidly adopting a more reproducible approach to data acquisition and analysis. Data structures and formats are being standardised and data analyses are getting more automated. However, as data analysis becomes more complicated, researchers often have to write longer analysis scripts, spanning different tools across multiple programming languages. This makes it more difficult to share or recreate one's code, harming the reproducibility of the analysis. We present a tool, Porcupine, that constructs one's analysis visually and automatically produces analysis code. The graphical representation improves understanding of the performed analysis, while retaining the flexibility of modifying the produced code manually to custom needs. Not only does Porcupine produce the analysis code, it also creates a shareable environment for running the code, in the form of a Docker image. Together, this forms a reproducible way of constructing, visualising and sharing one's analysis. Currently, Porcupine links to Nipype functionalities, which in turn accesses most standard neuroimaging tools. With Porcupine, we intend to bridge the gap between a conceptual and an implementational level of the analysis and thus create better reproducible and shareable science. We give the researcher a better oversight of their pipeline, both while developing and communicating their work. This may also be a useful tool in the education of novice neuroimaging students, considerably cutting their learning phase. We provide a wide range of examples and documentation, as well as installer files for all platforms on our website: https://timvanmourik.github.io/Porcupine.

Keywords: fMRI analysis, neuroimaging, visual programming

1. Introduction

The field of neuroimaging is rapidly adopting a more reproducible approach to data acquisition and analysis. Especially in recent years, a strong movement for doing better documented and more reproducible science can be observed. Advancements have been made in terms of openly sharing data (e.g. OpenFmri, (Poldrack et al., 2013)), standardizing data formats (BIDS format (Gorgolewski et al., 2016)), and facilitating more automated pipelines (Fischl et al., 2004; Gorgolewski et al., 2011; Jenkinson et al., 2012). These initiatives are great for increasing global scientific communication and collaboration, that is paramount in the age of big data.

*Corresponding author

However, as a result of the increasing complexity of analyses and the large scope of combining different tools, researchers often have to write custom scripts for combining different software packages, often in different programming languages. On top of this, many tools have external dependencies, intricate installation procedures, or different file formats for the same type of data. Additionally, the sharing initiatives usually have a stronger focus on sharing *data* (Human Connectome Project (Elam and Van Essen, 2015), NeuroVault (Gorgolewski et al., 2015)) instead of *code*, such that analysis scripts still have to be recreated based on the method section of a paper. All these factors greatly affect the reproducibility, documentation, and in the worst case correctness of the analysis (Brian A. Nosek, 2015).

Researchers maneuver themselves through these software limitations, but often the use of these pipelines assumes and requires a deeper level of understanding than is met by the proficiency of novice users. Specifically, scripting a pipeline that covers all low-level aspects of one's analysis without losing sight of the high-level aspects is daunting and error prone. Because of the slow learning curve associated with these sophisticated tools, there is the risk that novice users will revert to 'hacking' their analysis pipelines, sacrificing a reproducible approach. So as a novice user, how do you start an analysis? It is easiest to start with visualising the steps of your analysis pipeline.

In an increasingly complicated analysis environment there is a strong need for tools that give a better oversight of these complex analyses, while retaining the flexibility of combining different tools. A notable effort to integrate different tools is Nipype (Gorgolewski et al., 2011), that has a consistent Python interface to existing tools from all major MRI analysis packages. However, this still requires non-trivial Python scripting. Furthermore, Nipype is only able to visualise a workflow after it has been manually scripted (Ellson et al., 2002).

Here we detail our solution to these problems, an open-source software program we call Porcupine: 'PORcupine Creates Ur PipelINE - the worst recursive acronym with bad capitalisation and annoying use of slang.' Porcupine allows for the creation of neuroimaging pipelines by means of a graphical user interface (GUI). After graphical pipeline definition, Porcupine in turn creates the code that programmatically defines the pipeline. Additionally and without any additional overhead, we supply a Dockerfile (`https://www.docker.com`) that automatically builds the run environment for the pipeline. This not only facilitates sharing the pipeline, but also ensures its reproducibility (Boettiger, 2015). We provide an extensive list of examples and documentation on our website, as well as the possibility to upload one's custom pipeline to create a community driven library of analysis files.

By implementing an intermediate visual step in the generation of preprocessing workflows, Porcupine allows the user to focus on the logical flow and graph of the preprocessing pipeline without the need for coding at this conceptual stage of development. Because the GUI outputs working pipelining software code the user can then immediately inspect and alter the generated code. Thus, Porcupine provides a stepping stone that eases the transition from concept to implementation. Because the entire pipeline and its parameters are defined *in abstracto* before it is run, systems such as Nipype allow for elaborate checks and optimisations of the pipeline's execution. Furthermore, such systems can straightforwardly incorporate full logging of all analysis steps, creating a paper trail of the pipeline's execution. This combination of a reproducible environment in which a predefined pipeline is run by means of a system that provides precise bookkeeping paves the way to new standard that will ensure steady and reproducible progress in the field of cognitive neuroimaging (Gorgolewski and Poldrack, 2016).

In our practical experience, the use of Porcupine allows one to very quickly prototype preprocessing pipelines. Novice users can create a pipeline *de novo* and quickly focus on the code for this pipeline, greatly speeding up the learning process and thereby facilitating the use of reproducible pipelines. We envisage Porcupine to play a role in both the education of novice neuroimaging students and the rapid prototyping of pipelines by expert users. Here, we first outline several Porcupine use-case scenarios of

increasing complexity, after which we detail the architecture of Porcupine.

2. Results

2.1. What is Porcupine?

Porcupine is a graphical workflow editor that automatically produces analysis code from a graphically composed pipeline. By dropping 'nodes' (representing analysis steps) into the workflow editor and by connecting their data inputs and outputs, a pipeline is constructed. Analysis code is then automatically generated from the graphical representation of the pipeline. The code can readily be saved to a script (e.g. a Python, MATLAB, or Docker file) in order to perform the desired analysis. Additionally, the pipeline can be shared or inspected in visual form (PDF/SVG), or saved to a Porcupine specific (.pork) file to continue working on the pipeline at another time.

Apart from the visual representation of the pipeline, we provide more functionality to more orderly structure one's analysis, as outlined in Fig. 1. All functions (the nodes in the graph) that are included in the pipeline are also listed in a separate panel, listing their input parameters, output data, as well as a link to the online documentation of the function. We also provide the option to iterate over any input variable to facilitate parallelisation over subjects, sessions, or other variables. All parameters may also be edited in a separate parameter panel of the user interface. This functions as a central storage for important parameters, for example the ones that should be reported in a methods section. Porcupine combines the graphical overview and the parameters to automatically create the analysis code shown in the code editor.

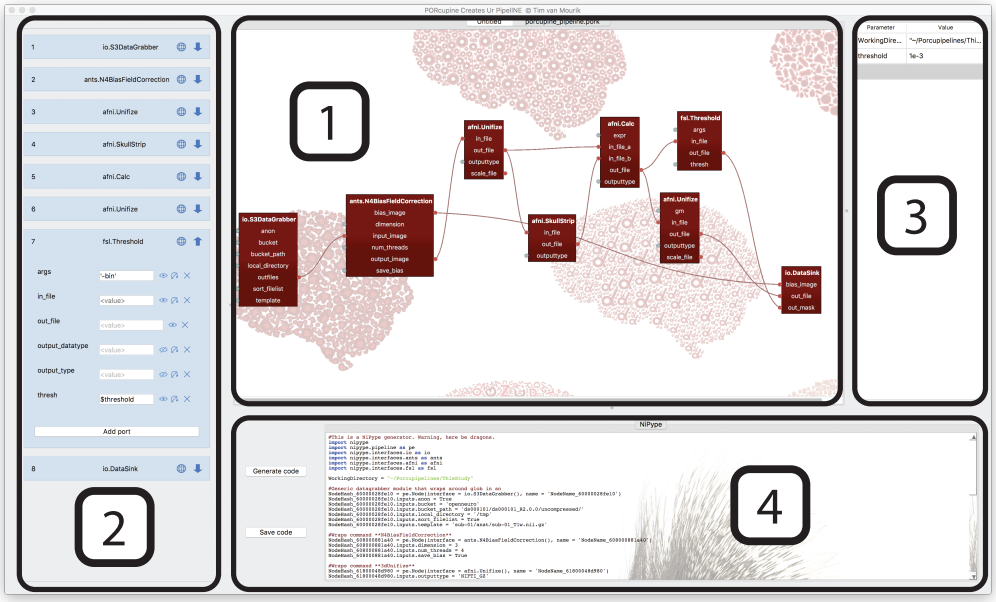

Figure 1: A screenshot of a Porcupine workflow. The editor is divided into four panels, each of them targeted at facilitating a more understandable and reproducible analysis. The *workflow editor* (1) provides a visual overview of one's analysis. The functions are all listed in the *node editor* (2), where the parameters for all functions can be orderly stored. This may include links to important parameters that are listed in the *parameter editor* (3), such that an overview of the main analysis settings can be easily viewed and modified. Readily executable analysis code is generated in the *code editor* (4)

We here focus on code generation that strictly adheres to the Nipype API (Gorgolewski et al., 2011), a Python-based MRI analysis and pipelining package. Nipype is used for its strong focus on uniformity in accessing functions, its link to most major MRI analysis tools, and its emphasis on reproducible science. Porcupine's architecture, however, is in principle agnostic with respect to the specific implementation of the underlying pipelining software. Any package with a consistent interface in the field of e.g. neuroimaging, bioengineering, or astronomy could benefit from using Porcupine's architecture.

We first show that we can easily generate a standard fMRI analysis pipeline. After visually dragging and dropping modules, code is automatically created that is customarily scripted manually instead. We then show how we facilitate loading data from an online repository, generate a readily executable fMRI pipeline, but also generate a shareable and reproducible analysis environment (using Docker), all with minimal additional effort. This allows for easily scalable analyses that could be performed locally, but also on computational clusters or with cloud computing, without manual installation of different software packages.

2.2. *Usage example*

We here show a simple example that constructs a pipeline for a single operation. In three steps, data is loaded, (minimally) processed, and the output is written to disk, as shown in Fig. 2. We here show an example that links to an OpenNeuro fMRI data set, but we could load any online data set that is set up according to the BIDS format (Gorgolewski et al., 2016). This Amazon repository ('S3 bucket') can be loaded by dragging the appropriate module into the workflow editor and typing the name of the bucket into the node editor. Its output can subsequently be connected to a Nipype function node, for example FSL's Brain Extraction Tool. All parameters of the function are listed and can be set in two different ways: either by dragging a link from a previous node's output port to an input port in the next node, or by typing in the parameter in the node editor. Subsequently, output can be written to disk by connecting the desired output to a Nipype DataSink node that collects and stores the data. By pressing the 'Generate code' button, the code for this pipeline is automatically generated and can immediately be saved and executed in a Python shell.

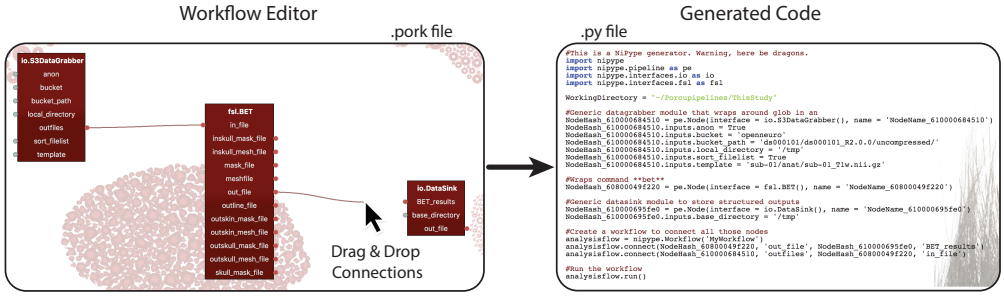

Figure 2: An example of simple workflow. In three steps, this pipeline loads data, processes it, and writes it to disk. This is achieved by connecting the input and output fields from subsequent nodes in the pipeline. The constructed workflow is then transformed in readily executable (Nipype) analysis code.

2.3. *Pipeline sharing*

From a simple example that reads and writes the data, a more complicated pipeline is readily set up. More functionality, i.e. nodes, can be dragged in and connected to quickly build a custom pipeline. As it is commonplace to repeat a single analysis or function for several subjects, sessions, or other variables, every field can be flagged as an 'iterator' field. This facilitates looping over variables. Once the pipeline is set

up and the code is generated, Nypipe offers functionality to construct a visual pipeline graph from custom python code. In Porcupine's proposed use-case, this end point of a standard Nipype pipeline represents the starting point, as shown in Fig. 3. This allows the user to focus on the desired pipeline graph first, and then progress to the manual editing of the generated code.

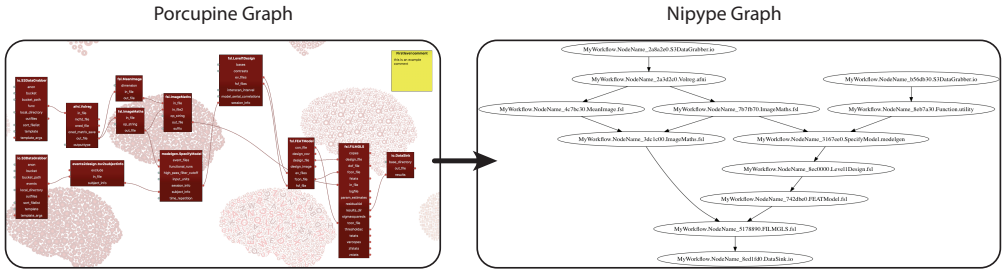

Figure 3: An example of a more complicated and realistic fMRI preprocessing pipeline. Once the code is generated, this can in turn be transformed into a Nipype graph visualisation. Whereas this is usually the end point for a pipeline in Nipype, we here propose to use a visualisation as a starting point of one's analysis.

Not only does Porcupine provide a way of setting up a preprocessing or analysis pipeline, we also provide a means for executing these pipelines in a reproducible environment. In addition to the Python analysis file that is generated, we create a scaffold for a Docker file. Docker (https://www.docker.com) is an open platform to easily build, run and share applications. The generated Docker file describes a minimal operating system that is required to run the analysis, depending on the dependencies of the modules used in the workflow editor. With this Docker file, an image of the full analysis can be built, shared and executed. This provides a simple platform to reproduce results of one's analysis, on the same data set, or on another with only a single change in the data source module. Alternatively, one can use it as a template environment for a new follow-up analysis. As with all generated code, the Docker code is fully customisable to a researcher's need, but our suggested scaffold requires only a single manual edit to be built as a Docker image: the addition of the analysis Python file.

```
ADD /path/to/pipeline/script.py /somewhere/porcupipeline.py
CMD ["python", "/somewhere/porcupipeline.py"]
```

Once this line is added, the docker image can be built:

```
$ docker build -t mydockerimage -f Dockerfile
```

The output from the executed pipeline can be written to a local directory on a researcher's computer ('/my/local/directory') by mounting it to the docker output directory ('/data') with the '-v' option and running the image as if it were a standalone application.

```
$ docker run -v /my/local/directory:/data mydockerimage
```

This docker command will execute the pipeline: load the data from an online repository, process the data, and store only the output data to a local directory. The docker image that includes the pipeline code as well as the run environment can be shared alongside a paper via DockerHub. The above examples (and many more) as well as extensive documentation and tutorials can be found here.

2.4. Limitations

Porcupine development is ongoing. This means that some features in Nipype have not been implemented. Notably, the JoinNode functionality is not yet accessible from the Porcupine user interface, in which the results from an upstream iterator are aggregated to a single output. Furthermore, custom extensions of Nipype functions are not automatically supported, but we do provide a script to add one's own custom module to Porcupine that would make this functionality accessible. A GUI-based solution for this is still a point of active discussion. In general, we encourage people to contribute new functionality in terms of modules, new toolboxes, and, most importantly, custom pipelines that can be added to the repository. Details for contributing can be found on the website.

It should be noted that Porcupine is not meant for low-level functionality, such as file handling and direct data operations. It is advised that the user either packages this into a custom module, or manually adds it to the produced code. Another limitation could be that Porcupine itself functions only as the front-end to the Nipype back-end, but does not perform type-matching on the input and output of a connection, nor does it perform syntax checking of the manually edited parameters. As Nipype already performs these tasks, this was a deliberate division-of-labour decision.

3. Methods

Porcupine's graphical user interface was written first with a general visual programming application in mind. The initial interface to Nipype was developed at a three-day coding sprint at BrainHack 2017, Amsterdam. This kickstarted Porcupine in its current form. The source code, as well as the installer files for Windows, Mac, and Linux, are publicly available as a GitHub repository.

Visual programming is a generic way of programming to create a data flow or to perform an ordered task with a modular structure (Myers, 1986). Customarily, it allows the user to construct a Directed Acyclic Graph (DAG) (Thulasiraman and Swamy, 1992) of conceptualised operations that are subsequently interpreted or compiled as an application (Myers, 1990). This format is particularly useful for workflows that fit modular structures, such as most neuroimaging data analyses (Rex et al., 2003).

3.1. Architecture

Not only do we intend researchers to make their analyses (re-)usable and robust, our software also adheres to all 20 simple rules that were laid out to this end (List et al., 2017; Taschuk and Wilson, 2017). The updates as well as the releases of the source code are realised by means of a GitHub repository. Installer files are provided for all platforms and do not require administrator privilege. Users can get started quickly by extensive documentation and an example gallery.

Easy cross-platform installation or compilation was achieved by programming Porcupine as a stand-alone application in Qt Creator (`https://www.qt.io`) for C++. Internal file formats were standardised to JSON dictionaries, a format native to Python, Qt, and web applications. This provides a simple means to add new modules to Porcupine, without the need to write additional code. Every dictionary specifies a software package (e.g.'Nipype', 'Docker', etc.) that is interpreted by Porcupine and creates code that is native to the package. A package-specific interpreter needs to be written just once, after which new modules that are included in the dictionary will be automatically available in Porcupine.

Each JSON dictionary describes a list of functions (internally referred to as 'nodes'). Each function has a name and (optionally) a category, a web url to its documentation, and a block of code. A code block specifies the software package for which the node is meant, the associated piece of code for that function and optionally an additional comment. Furthermore, a node contains any number of data/parameter ports, each of which can be input, output, or both. Optionally, additional flags can be set for ports to be visible in

the editor, whether its value is editable, or whether the variable needs to be iterated over. Thus, JSON files for custom nodes can easily be created and added as a dictionary to the graphical interface. We also provide a Python script that converts one's custom Python function(s) to a Nipype node dictionary.

3.2. Extending Porcupine with new toolboxes

Currently, Porcupine features Nipype and Docker support, but this could easily be extended to other software packages. This requires no major changes to the Porcupine source code, merely the inclusion of a single C++ class that describes the relationship between the nodes, links, and the output code. Specifically, the 'CodeGenerator' class must be inherited and has access to the full workflow: the list of nodes, their parameters, and their connections. As long as all functions within an analysis toolbox can be accessed with a consistent interface, they can be represented as modules within Porcupine. This way, the developers of the Fastr framework programmed initial support for their code base (Achterberg et al., 2016). Unfortunately, only few neuroimaging abide by this uniformity of their functions and hence many cannot be included into Porcupine.

3.3. Relation to existing pipeline managers

Porcupine aims to provide an extendable, transparent and flexible platform to build preprocessing and analysis pipelines. Other software packages have made similar attempts at providing visual aids to build or run pipelines. Within neuroimaging, the most notable ones are CBS Tools (Bazin et al., 2014) and the LONI pipeline (Rex et al., 2003). Porcupine distinguishes itself from these by not creating a run environment, but instead creating the analysis code for the researcher. This retains the possibility of immediately running the code through a Python interpreter, but also creates more flexibility, as one can modify and adjust the script to one's personal need. Additionally, as the functions link to existing Nipype interfaces, error reporting is more transparent such that problems can more easily be resolved. Lastly, our open-source framework is set up to be extendable with new modules within existing frameworks, as well as with completely new frameworks. This provides a future-proof set-up for current and future analysis tools in neuroimaging and beyond.

4. Discussion

We have presented a new tool to visually construct an analysis pipeline. Subsequently, Porcupine automatically generates the analysis code, and provides a way of running and sharing such analyses. We see this as an important tool and a stepping stone on the path to doing more reproducible and open science. Additionally, this gives researchers a better oversight of their analysis pipeline, allowing for greater ease at developing, understanding, and communicating complex analyses.

Porcupine provides two independent functionalities that dovetail to allow users to more easily take part in reproducible neuroimaging research. They are (1) a graphical user interface for the visual design of analysis pipelines and (2) a framework for the automated creation of docker images to execute and share the designed analysis.

We anticipate that the ability to design processing pipelines visually instead of programmatically cuts the novice user's learning phase by a considerable amount of time by facilitating understanding and development. The ease of use of a Graphical User Interface (GUI) implementation extends and complements Nipype's flexibility. Thus, it invites researchers to mix and match different tools, and adhere less stringently to the exclusive use of the tools of any given toolbox ecosystem. This flexibility enhances the possible sophistication of processing pipelines, and could for instance be helpful in cross-modal research or multi-site research. Additionally, it may nudge method developers to write new tools in a way that easily integrates with the Nipype and Porcupine structure.

The emphasis that Porcupine puts on visual development of analyses makes it easier to communicate a methods section visually rather than in writing. We foresee that researchers may prefer explicity sharing the created .pork files and the Nipype pipelines that are created from them, instead of solely relying on written descriptions of their methods. Yet another use case for Porcupine is the easy definition of proposed processing workflows for preregistered studies.

Importantly, Porcupine attempts to improve the steepness of the learning curve that is inherent to the use of complex analysis, by providing a more structured and systematic approach to pipeline creation. This places Porcupine in a position to aid in the education of novice neuroimaging researchers, as it allows them to focus on the logic of their processing instead of the creation of the code for the processing - greatly improving and accelerating their understanding of the different steps involved in the preprocessing of neuroimaging data.

Having allowed for the visual design of a pipeline for the preprocessing or analysis of a neuroimaging dataset, the reproducible execution of this pipeline is another step that Porcupine facilitates. By flexibly creating a Docker image tailored to the different preprocessing steps defined visually in the GUI, Porcupine allows the user to share not only the definition of the pipeline but also its execution environment. This step removes the overhead of having to manually install the desired operating system with the matched distribution of MRI analysis software. This final step greatly facilitates the reproducibility of reported results, and is part of a general evolution of the field towards easily shareable and repeatable analyses.

From this point, with only trivial additions to the Dockerfile, it can be transformed into a BIDS app (Gorgolewski et al., 2017), and providing a direct way of creating this is high on the feature list of Porcupine. This is explained in detail on our website. Additionally, integrating support for standardised workflow file formats, such as the Common Workflow Language (Amstutz et al., 2016), may be added. Another point of improvement is a functionality to embed pipelines within pipelines. Currently, a complicated pipelines does full justice to the term 'spaghetti code', and the number of nodes and links may easily compromise the visual aid in understanding; the very purpose for which Porcupine was created. This may easily be solved by compartmentalising pipelines into logical units by providing an embedded structure.

We intend Porcupine to be a strong aid for doing better, more reproducible and shareable science. By bridging the gap between a conceptual and implementational level of the analysis, we give scientists a better oversight of their pipeline and aid them in developing and communicating their work. We provide extensive and intuitive documentation and a wide range of examples to give users a frictionless start to use Porcupine. We look forward to adding more functionality and support for more toolboxes in the near future.

5. Author contributions

TvM wrote the Porcupine C++ software. TvM, LS, and TK designed the interface to Nipype. LS and TvM built the website. LS wrote the examples and the majority of documentation. TvM, TK, LS, and DGN wrote the paper.

6. Acknowledgements

We would like to thank the organisation of BrainHack Global and BrainHack Amsterdam, specifically Pierre-Louis Bazin, for organising the platform that kickstarted Porcupine in its current form.

7. References

Achterberg, H. C., Koek, M., Niessen, W. J., 2016. Fastr: A workflow engine for advanced data flows in medical image analysis. Frontiers in ICT 3, 15.
URL http://journal.frontiersin.org/article/10.3389/fict.2016.00015

Amstutz, P., Crusoe, M. R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., Leehr, D., Ménager, H., Nedeljkovich, M., et al., 2016. Common workflow language, v1. 0. figshare.

Bazin, P.-L., Weiss, M., Dinse, J., Schäfer, A., Trampel, R., Turner, R., 2014. A computational framework for ultra-high resolution cortical segmentation at 7tesla. Neuroimage 93, 201–209.

Boettiger, C., 2015. An introduction to docker for reproducible research. ACM SIGOPS Operating Systems Review 49 (1), 71–79.

Brian A. Nosek, O. S. C., 2015. Estimating the reproducibility of psychological science. Science 349 (6251).
URL http://science.sciencemag.org/content/349/6251/aac4716

Elam, J. S., Van Essen, D., 2015. Human Connectome Project. Springer New York, New York, NY, pp. 1408–1411.
URL https://doi.org/10.1007/978-1-4614-6675-8_592

Ellson, J., Gansner, E., Koutsofios, L., North, S. C., Woodhull, G., 2002. Graphviz— Open Source Graph Drawing Tools. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 483–484.
URL https://doi.org/10.1007/3-540-45848-4_57

Fischl, B., van der Kouwe, A., Destrieux, C., Halgren, E., Segonne, F., Salat, D. H., Busa, E., Seidman, L. J., Goldstein, J., Kennedy, D., Caviness, V., Makris, N., Rosen, B., Dale, A. M., Jan 2004. Automatically parcellating the human cerebral cortex. Cereb. Cortex 14 (1), 11–22.

Gorgolewski, K., Burns, C., Madison, C., Clark, D., Halchenko, Y., Waskom, M., Ghosh, S., 2011. Nipype: A flexible, lightweight and extensible neuroimaging data processing framework in python. Frontiers in Neuroinformatics 5, 13.
URL http://journal.frontiersin.org/article/10.3389/fninf.2011.00013

Gorgolewski, K. J., Alfaro-Almagro, F., Auer, T., Bellec, P., CapotÄČ, M., Chakravarty, M. M., Churchill, N. W., Cohen, A. L., Craddock, R. C., Devenyi, G. A., Eklund, A., Esteban, O., Flandin, G., Ghosh, S. S., Guntupalli, J. S., Jenkinson, M., Keshavan, A., Kiar, G., Liem, F., Raamana, P. R., Raffelt, D., Steele, C. J., Quirion, P.-O., Smith, R. E., Strother, S. C., Varoquaux, G., Wang, Y., Yarkoni, T., Poldrack, R. A., 03 2017. Bids apps: Improving ease of use, accessibility, and reproducibility of neuroimaging data analysis methods. PLOS Computational Biology 13 (3), 1–16.
URL https://doi.org/10.1371/journal.pcbi.1005209

Gorgolewski, K. J., Auer, T., Calhoun, V. D., Craddock, R. C., Das, S., Duff, E. P., Flandin, G., Ghosh, S. S., Glatard, T., Halchenko, Y. O., Handwerker, D. A., Hanke, M., Keator, D., Li, X., Michael, Z., Maumet, C., Nichols, B. N., Nichols, T. E., Pellman, J., Poline, J.-B., Rokem, A., Schaefer, G., Sochat, V., Triplett, W., Turner, J. A., Varoquaux, G., Poldrack, R. A., Jun. 2016. The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments 3, 160044.
URL http://dx.doi.org/10.1038/sdata.2016.44

Gorgolewski, K. J., Poldrack, R. A., Jul. 2016. A practical guide for improving transparency and reproducibility in neuroimaging research. PLoS Biology 14 (7), e1002506.
URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4936733/

Gorgolewski, K. J., Varoquaux, G., Rivera, G., Schwarz, Y., Ghosh, S. S., Maumet, C., Sochat, V. V., Nichols, T. E., Poldrack, R. A., Poline, J.-B., Yarkoni, T., Margulies, D. S., 2015. Neurovault.org: a web-based repository for collecting and sharing unthresholded statistical maps of the human brain. Frontiers in Neuroinformatics 9, 8.
URL http://journal.frontiersin.org/article/10.3389/fninf.2015.00008

Jenkinson, M., Beckmann, C. F., Behrens, T. E., Woolrich, M. W., Smith, S. M., 2012. Fsl. Neuroimage 62 (2), 782–790.

List, M., Ebert, P., Albrecht, F., 01 2017. Ten simple rules for developing usable software in computational biology. PLOS Computational Biology 13 (1), 1–5.
URL https://doi.org/10.1371/journal.pcbi.1005265

Myers, B. A., Apr. 1986. Visual programming, programming by example, and program visualization: A taxonomy. SIGCHI Bull. 17 (4), 59–66.
URL http://doi.acm.org/10.1145/22339.22349

Myers, B. A., 1990. Taxonomies of visual programming and program visualization. Journal of Visual Languages & Computing 1 (1), 97 – 123.
URL http://www.sciencedirect.com/science/article/pii/S1045926X05800369

Poldrack, R., Barch, D., Mitchell, J., Wager, T., Wagner, A., Devlin, J., Cumba, C., Koyejo, O., Milham, M., 2013. Toward open sharing of task-based fmri data: the openfmri project. Frontiers in Neuroinformatics 7, 12.
URL http://journal.frontiersin.org/article/10.3389/fninf.2013.00012

Rex, D. E., Ma, J. Q., Toga, A. W., 2003. The loni pipeline processing environment. NeuroImage 19 (3), 1033 – 1048.
URL http://www.sciencedirect.com/science/article/pii/S105381190300185X

Taschuk, M., Wilson, G., 04 2017. Ten simple rules for making research software more robust. PLOS Computational Biology 13 (4), 1–10.
URL https://doi.org/10.1371/journal.pcbi.1005412

Thulasiraman, K., Swamy, M., 1992. 5.7 acyclic directed graphs. Graphs: Theory and Algorithms, 118.