RESEARCH ARTICLE

Porcupine: A visual pipeline tool for neuroimaging analysis

Tim van Mourik[1], **Lukas Snoek**[2], **Tomas Knapen**[3,4], **David G. Norris**[1,5]

1 Radboud University Nijmegen, Donders Institute for Brain, Cognition and Behaviour, Kapittelweg, Nijmegen, The Netherlands, **2** University of Amsterdam, Department of Brain & Cognition, Nieuwe Achtergracht, Amsterdam, The Netherlands, **3** Cognitive Psychology & Institute for Brain & Behavior, Vd Boechorststraat, Amsterdam, The Netherlands, **4** Spinoza Centre for Neuroimaging, Meibergdreef, Amsterdam, The Netherlands, **5** Erwin L. Hahn Institute for Magnetic Resonance Imaging, Arendahls Wiese, Essen, Germany

* t.vanmourik@donders.ru.nl

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 code, reducing 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 analysis tools. Our goal is to release researchers from the constraints of specific implementation details, thereby freeing them to think about novel and creative ways to solve a given problem. Porcupine improves the overview researchers have of their processing pipelines, and facilitates both the development and communication of their work. This will reduce the threshold at which less expert users can generate reusable pipelines. With Porcupine, we bridge the gap between a conceptual and an implementational level of analysis and make it easier for researchers to create reproducible and shareable science. 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. Porcupine is free, open source, and released under the GNU General Public License v3.0.

This is a *PLOS Computational Biology* Software paper.

Citation: van Mourik T, Snoek L, Knapen T, Norris DG (2018) Porcupine: A visual pipeline tool for neuroimaging analysis. PLoS Comput Biol 14(5): e1006064. https://doi.org/10.1371/journal.pcbi.1006064

Editor: Andreas Prlic, UCSD, UNITED STATES

Received: October 23, 2017

Accepted: February 27, 2018

Published: May 10, 2018

Data Availability Statement: All software, documentation, and examples may be found on https://timvanmourik.github.io/Porcupine.

Funding: Tim van Mourik acknowledges support from the Royal Netherlands Academy of Arts and Sciences, from the Academy Professor Prize 2012, awarded to professor Peter Hagoort. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing interests: The authors have declared that no competing interests exist.

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 conducting better documented and more reproducible science can be observed. Advances have been made in terms of openly sharing data (e.g. OpenFmri, [1]), standardizing data formats (BIDS format [2]), and facilitating more automated pipelines [3–5]. These initiatives facilitate increasing global scientific communication and collaboration, that is paramount in the age of big data.

As a result of the increasing complexity of analyses and the wide variety of different tools, researchers often have to write custom scripts for combining different software packages, often in different programming languages. As an extra obstacle, many tools have external dependencies, intricate installation procedures, or different file formats for the same type of data. Furthermore, the sharing initiatives usually have a stronger focus on sharing *data* (Human Connectome Project [6], NeuroVault [7]) instead of *code*, such that analysis scripts still have to be recreated based on the method section of a paper. All these factors negatively affect the reproducibility, documentation, and in the worst case correctness of the analysis [8].

A considerable mastery of coding is required for analysing fMRI data. The conceptual side of understanding all preprocessing steps is not trivial, but converting this into a working pipeline can be an arduous journey. The necessary programming skills are not usually the prime focus of a brain researcher's skills or interests, but they are a necessity for completing one's analysis. Consequently, scripting a pipeline that covers all high-level and low-level aspects is daunting and error prone. As a result, there is a considerable risk of 'hacking' an analysis pipeline together, sacrificing a reproducible approach. So as a researcher, 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 [4], that has a 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 [9].

Here we detail our solution to these problems, an open-source software program we call Porcupine: 'PORcupine Creates Ur PipelINE. Porcupine allows 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 [10]. 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 analyses.

By implementing an intermediate visual step in the generation of preprocessing workflows, Porcupine allows the user to focus on the logical flow of the preprocessing pipeline in a graphical representation without the need for coding at this conceptual stage of development. Because the GUI produces functional analysis code, the user can then immediately inspect, save, and run 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 [11].

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.

Results

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.

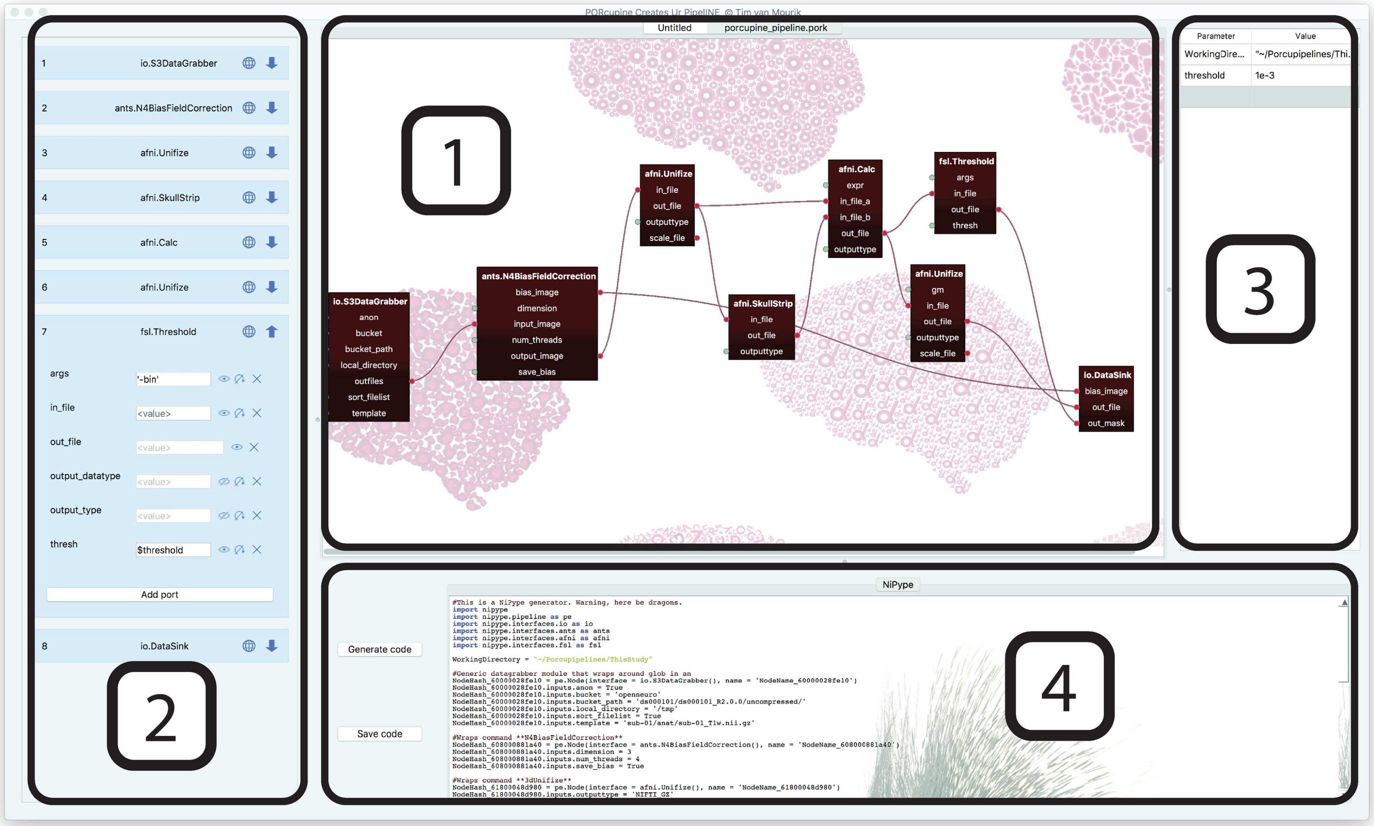
Apart from the visual representation of the pipeline, we provide more functionality to orderly structure one's analysis, as outlined in Fig 1. All functions (the nodes in the graph) that are included in the pipeline are also listed in a separate panel, listing their input parameters, output data, as well as a link to the online documentation of the function. We also provide the option to iterate over any input variable in order 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 window.

We here focus on code generation that strictly adheres to the Nipype API [4], 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 usually 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 can be performed locally, but also on computational clusters or with cloud computing, without manual installation of different software packages.

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



1	io.S3DataGrabber
2	ants.N4BiasFieldCorrection
3	afni.Unifize
4	afni.SkullStrip
5	afni.Calc
6	afni.Unifize
7	fsl.Threshold
8	io.DataSink

args	'-bin'
in_file	<value>
out_file	<value>
output_datatype	<value>
output_type	<value>
thresh	$threshold

Parameter	Value
WorkingDire...	"~/Porcupipelines/Thi...
threshold	1e-3

```
#This is a NiPype generator. Warning, here be dragons.
import nipype
import nipype.pipeline as pe
import nipype.interfaces.io as io
import nipype.interfaces.ants as ants
import nipype.interfaces.afni as afni
import nipype.interfaces.fsl as fsl

WorkingDirectory = "~/Porcupipelines/ThisStudy"

#Generic datagrabber module that wraps around glob in an
NodeHash_60000028fe10 = pe.Node(interface = io.S3DataGrabber(), name = 'NodeName_60000028fe10')
NodeHash_60000028fe10.inputs.anon = True
NodeHash_60000028fe10.inputs.bucket = 'openneuro'
NodeHash_60000028fe10.inputs.bucket_path = 'ds000101/ds000101_R2.0.0/uncompressed/'
NodeHash_60000028fe10.inputs.local_directory = '/tmp'
NodeHash_60000028fe10.inputs.sort_filelist = True
NodeHash_60000028fe10.inputs.template = 'sub-01/anat/sub-01_T1w.nii.gz'

#Wraps command **N4BiasFieldCorrection**
NodeHash_608000881a40 = pe.Node(interface = ants.N4BiasFieldCorrection(), name = 'NodeName_608000881a40')
NodeHash_608000881a40.inputs.dimension = 3
NodeHash_608000881a40.inputs.num_threads = 4
NodeHash_608000881a40.inputs.save_bias = True

#Wraps command **3dUnifize**
NodeHash_61800048d980 = pe.Node(interface = afni.Unifize(), name = 'NodeName_61800048d980')
NodeHash_61800048d980.inputs.outputtype = 'NIFTI_GZ'
```

Fig 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 window* (4).

https://doi.org/10.1371/journal.pcbi.1006064.g001

data set that is set up according to the BIDS format [2]. OpenNeuro's data sets are stored as Amazon repositories ('S3 buckets') and 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

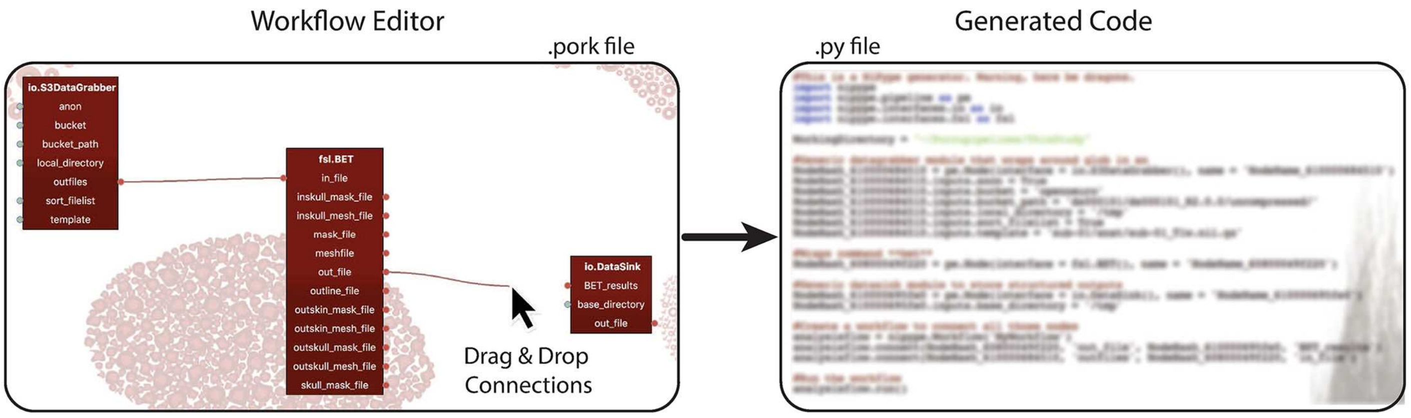


Fig 2. An example of simple workflow. In three steps, this pipeline loads data, processes it, and writes it to disk. This is achieved by connecting the input and output fields from subsequent nodes in the pipeline. The constructed workflow is then transformed in readily executable (Nipype) analysis code.

https://doi.org/10.1371/journal.pcbi.1006064.g002

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.

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, Nipype 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.

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, based 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 (see S1 File. Docker files). The 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 includes both the pipeline code and the run environment, and

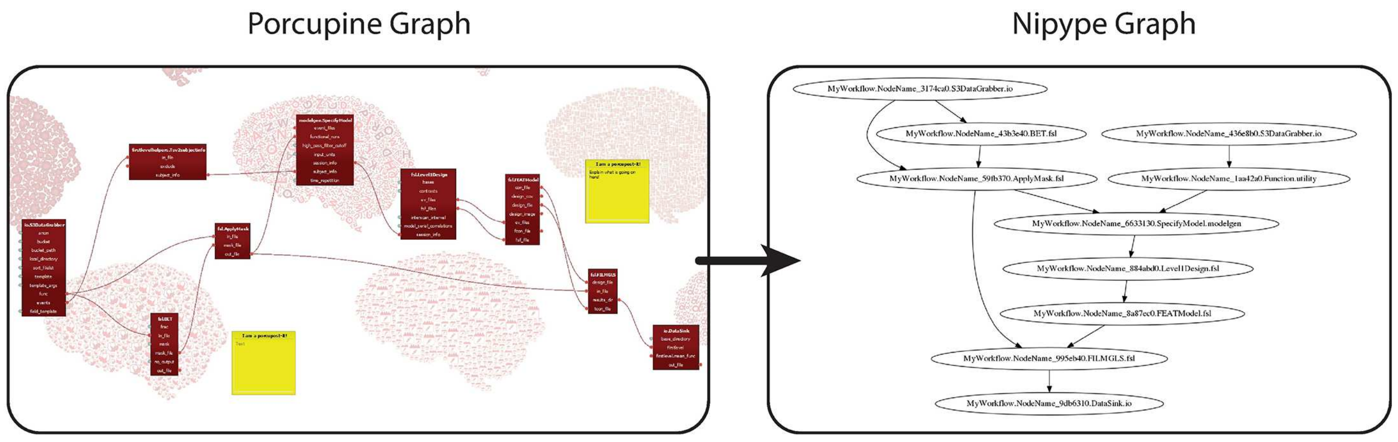

Porcupine Graph
Nipype Graph
MyWorkflow.NodeName_3174ca0.S3DataGrabber.io
MyWorkflow.NodeName_43b3e40.BET.fsl
MyWorkflow.NodeName_436e8b0.S3DataGrabber.io
MyWorkflow.NodeName_59fb370.ApplyMask.fsl
MyWorkflow.NodeName_1aa42a0.Function.utility
MyWorkflow.NodeName_6633130.SpecifyModel.modelgen
MyWorkflow.NodeName_884abd0.Level1Design.fsl
MyWorkflow.NodeName_8a87ec0.FEATModel.fsl
MyWorkflow.NodeName_995eb40.FILMGLS.fsl
MyWorkflow.NodeName_9db6310.DataSink.io

Fig 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.

https://doi.org/10.1371/journal.pcbi.1006064.g003 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.

Limitations

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 for this is still an intended point of improvement. In general, feature requests are maintained as *issues* and *projects* in the GitHub repository. We encourage people to contribute new ideas or implementations for functionality in terms of modules, new toolboxes, and, most importantly, custom pipelines that can be added to the repository. Details on how to contribute can be found on the website.

While Porcupine in principle supports all workflow operations, a specific pipeline may well require modules that are not provided within Nipype. It is advised that the user either packages custom code for this into a new module, or manually adds it to the produced code. We furthermore stress that Porcupine is intended to function as a front-end encapsulation of NiPype, and does not implement the parsing of python files that contain pre-defined nipype pipelines. It also does not perform type-matching on the input and output of a connection, nor does it perform syntax checking of the manually edited parameters.

Design and implementation

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. Porcupine is free, open source, and released under the GNU General Public License v3.0. It has static digital object identifier (DOI) doi.org/10.5281/zenodo.1146653.

Visual programming is a generic way of programming to create a data flow or to perform an ordered task with a modular structure [12]. Customarily, it allows the user to construct a Directed Acyclic Graph (DAG) [13] of conceptualised operations that are subsequently interpreted or compiled as an application [14]. This format is particularly useful for workflows that fit modular structures, such as most neuroimaging data analyses [15].

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 [16, 17]. 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 are aided in getting 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 a custom Python function(s) to a Nipype node dictionary.

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. Apart from Nipype, support for a laminar specific fMRI analysis toolbox in MATLAB is provided. The developers of the Fastr framework programmed initial support for their code base [18]. Unfortunately, only few neuroimaging packages abide by this uniformity of their functions and hence many cannot be included into Porcupine.

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 the JIST pipeline [19], extended with CBS Tools [20] and the LONI pipeline [15]. 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 researchers can modify and adjust the script according to their needs. 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 perhaps other disciplines.

Availability and future directions

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 of 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 will cut 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 reduce 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. It separates the skill of building a conceptual analysis pipeline from the skill of coding this in the appropriate programming language. 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. At the same time, it allows more experienced researchers to spend more time on the conceptual side than on implementational side.

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 matching 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.

The generated Docker image can be made High Performance Computing aware by means of dedicated tools such as docker2singularity. Alternatively, with only trivial additions to the Dockerfile, it can be transformed into a BIDS app [21]. A detailed explanation for doing this can be found on our website. An automatic and direct way of creating this has not yet been implemented. Additionally, integrating support for standardised workflow file formats, such as the Common Workflow Language [22] could further add to Porcupine's aim of reproducibility. Another point of improvement is a functionality to embed pipelines within pipelines. Currently, a complicated pipeline 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 supporting more toolboxes in the near future.

Supporting information

S1 File. Docker files.
(PDF)

Acknowledgments

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.

Author Contributions

Conceptualization: Tim van Mourik.

Investigation: Tim van Mourik.

Methodology: Tim van Mourik, Tomas Knapen, David G. Norris.

Project administration: Tim van Mourik.

Resources: Tim van Mourik, Lukas Snoek.

Software: Tim van Mourik, Lukas Snoek.

Supervision: David G. Norris.

Validation: Lukas Snoek.

Visualization: Tim van Mourik.

Writing – original draft: Tim van Mourik, Lukas Snoek, Tomas Knapen.

Writing – review & editing: Tim van Mourik, Lukas Snoek, David G. Norris.

References

1. Poldrack R, Barch D, Mitchell J, Wager T, Wagner A, Devlin J, et al. Toward open sharing of task-based fMRI data: the OpenfMRI project. Frontiers in Neuroinformatics. 2013; 7:12. https://doi.org/10.3389/fninf.2013.00012 PMID: 23847528
2. Gorgolewski KJ, Auer T, Calhoun VD, Craddock RC, Das S, Duff EP, et al. The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments. 2016; 3:160044.
3. Fischl B, van der Kouwe A, Destrieux C, Halgren E, Segonne F, Salat DH, et al. Automatically parcellating the human cerebral cortex. Cereb Cortex. 2004; 14(1):11–22. https://doi.org/10.1093/cercor/bhg087 PMID: 14654453
4. Gorgolewski K, Burns C, Madison C, Clark D, Halchenko Y, Waskom M, et al. Nipype: A Flexible, Lightweight and Extensible Neuroimaging Data Processing Framework in Python. Frontiers in Neuroinformatics. 2011; 5:13. https://doi.org/10.3389/fninf.2011.00013 PMID: 21897815
5. Jenkinson M, Beckmann CF, Behrens TE, Woolrich MW, Smith SM. Fsl. Neuroimage. 2012; 62 (2):782–790. https://doi.org/10.1016/j.neuroimage.2011.09.015 PMID: 21979382
6. Elam JS, Van Essen D. In: Jaeger D, Jung R, editors. Human Connectome Project. New York, NY: Springer New York; 2015. p. 1408–1411.
7. Gorgolewski KJ, Varoquaux G, Rivera G, Schwarz Y, Ghosh SS, Maumet C, et al. NeuroVault.org: a web-based repository for collecting and sharing unthresholded statistical maps of the human brain. Frontiers in Neuroinformatics. 2015; 9:8. https://doi.org/10.3389/fninf.2015.00008 PMID: 25914639
8. Brian A, Nosek OSC. Estimating the reproducibility of psychological science. Science. 2015; 349 (6251). https://doi.org/10.1126/science.aac4716
9. Ellson J, Gansner E, Koutsofios L, North SC, Woodhull G. In: Mutzel P, Jünger M, Leipert S, editors. Graphviz— Open Source Graph Drawing Tools. Berlin, Heidelberg: Springer Berlin Heidelberg; 2002. p. 483–484.

10. Boettiger C. An introduction to Docker for reproducible research. ACM SIGOPS Operating Systems Review. 2015; 49(1):71–79. https://doi.org/10.1145/2723872.2723882

11. Gorgolewski KJ, Poldrack RA. A Practical Guide for Improving Transparency and Reproducibility in Neuroimaging Research. PLoS Biology. 2016; 14(7):e1002506. https://doi.org/10.1371/journal.pbio.1002506 PMID: 27389358

12. Myers BA. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. SIGCHI Bull. 1986; 17(4):59–66. https://doi.org/10.1145/22339.22349

13. Thulasiraman K, Swamy M. 5.7 Acyclic Directed Graphs. Graphs: Theory and Algorithms. 1992; p. 118.

14. Myers BA. Taxonomies of visual programming and program visualization. Journal of Visual Languages & Computing. 1990; 1(1):97–123. http://dx.doi.org/10.1016/S1045-926X(05)80036-9.

15. Rex DE, Ma JQ, Toga AW. The LONI Pipeline Processing Environment. NeuroImage. 2003; 19 (3):1033–1048. http://dx.doi.org/10.1016/S1053-8119(03)00185-X.

16. List M, Ebert P, Albrecht F. Ten Simple Rules for Developing Usable Software in Computational Biology. PLOS Computational Biology. 2017; 13(1):1–5. https://doi.org/10.1371/journal.pcbi.1005265

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

18. Achterberg HC, Koek M, Niessen WJ. Fastr: A Workflow Engine for Advanced Data Flows in Medical Image Analysis. Frontiers in ICT. 2016; 3:15. https://doi.org/10.3389/fict.2016.00015

19. Lucas BC, Bogovic JA, Carass A, Bazin PL, Prince JL, Pham DL, et al. The Java Image Science Toolkit (JIST) for Rapid Prototyping and Publishing of Neuroimaging Software. Neuroinformatics. 2010; 8(1):5–17. https://doi.org/10.1007/s12021-009-9061-2 PMID: 20077162

20. Bazin PL, Weiss M, Dinse J, Schäfer A, Trampel R, Turner R. A computational framework for ultra-high resolution cortical segmentation at 7Tesla. Neuroimage. 2014; 93:201–209. https://doi.org/10.1016/j.neuroimage.2013.03.077 PMID: 23623972

21. Gorgolewski KJ, Alfaro-Almagro F, Auer T, Bellec P, Capotă M, Chakravarty MM, et al. BIDS apps: Improving ease of use, accessibility, and reproducibility of neuroimaging data analysis methods. PLOS Computational Biology. 2017; 13(3):1–16. https://doi.org/10.1371/journal.pcbi.1005209

22. Amstutz P, Crusoe MR, Tijanić N, Chapman B, Chilton J, Heuer M, et al. Common Workflow Language, v1. 0. figshare. 2016;.