A Graphical Application to Visualize Nipype Workflows

Blake Jackson ‘16
05.15.2016

Supervisor Satrajit Ghosh
6.UAP Project

Abstract

Nipype is a python project that allows diverse neuroimaging software to interact under a single workflow. The output of Nipype is stored as a provenance graph, where nodes represent algorithmic processes, and connections between nodes represent the flow of data between these processes. This report describes **Workflow Viewer**, a web application specifically designed to graphically display these workflow graphs and allow for their efficient navigation.

Overview

Due to the rapid expansion of neuroscience research, several neuroimaging applications have come about that provide highly developed processing algorithms, often supporting similar functionality. However, these applications are often coded on diverse platforms (e.g., Python, MATLAB, Java, command-line executables) and lack a uniform operating interface. Neuroscientists, who often don't have much coding experience, are forced to learn how to interoperate between these applications or simply stick to one platform.

Nipype[i] eliminates this need by creating a uniform Python interface across neuroimaging applications and providing a scriptable mechanism to combine applications into workflows. Workflows are represented as directed acyclic graphs, where each node represents a process that takes in specific inputs and computes outputs. Nodes can be combined in such a way to create complex processes and produce valuable output. When a workflow is ready to be run, it is

transformed into an execution graph and executed. The output of this execution is stored as a provenance graph using the W3C PROV standard[ii].

Many of the users who have a need for Nipype are unfamiliar with Python or with provenance data structures, and it is therefore important that Nipype offer a graphical interface that allows for the visualization and manipulation of the provenance graphs that Nipype outputs. Workflow Viewer is a web application that accomplishes this purpose. Given a workflow graph, Workflow Viewer displays a topologically sorted table of the nodes in the graph. To avoid clutter, no arrows are displayed between nodes; instead, a user can investigate connections between nodes by clicking on any particular node in which he or she is interested. When a node is clicked, it expands, and the node's parents and children are colored differently in the table. Because the graph is topologically sorted, all of the node's parents are displayed above the node, and all of the node's children are displayed below.

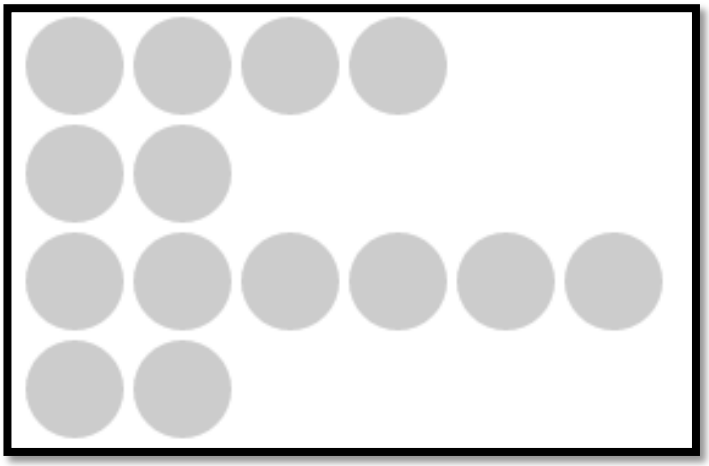

Figure 1: A simple workflow graph displayed in Workflow Viewer.

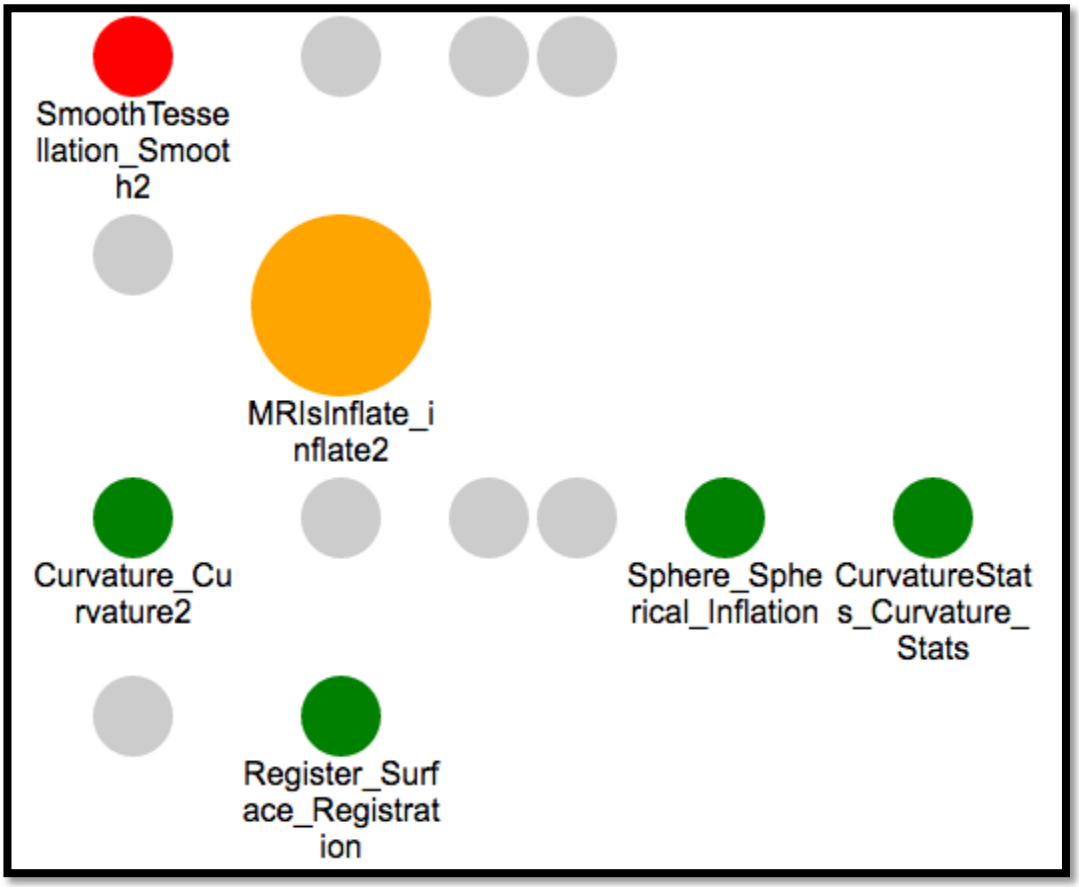

Figure 2: The same workflow graph expanded about one of its nodes. The selected node is colored orange, the node's parents are colored red, and the node's children are colored green.

A node also contains inputs and outputs, but these are not displayed in the graph table. Instead, they are displayed in an information panel at the bottom of the page. When a user hovers over or clicks one of these inputs or outputs, the specific node in the graph that the input/output feeds into lights up. This allows users to know exactly what algorithmic process (node) is contributing to a specific input of any node in question.

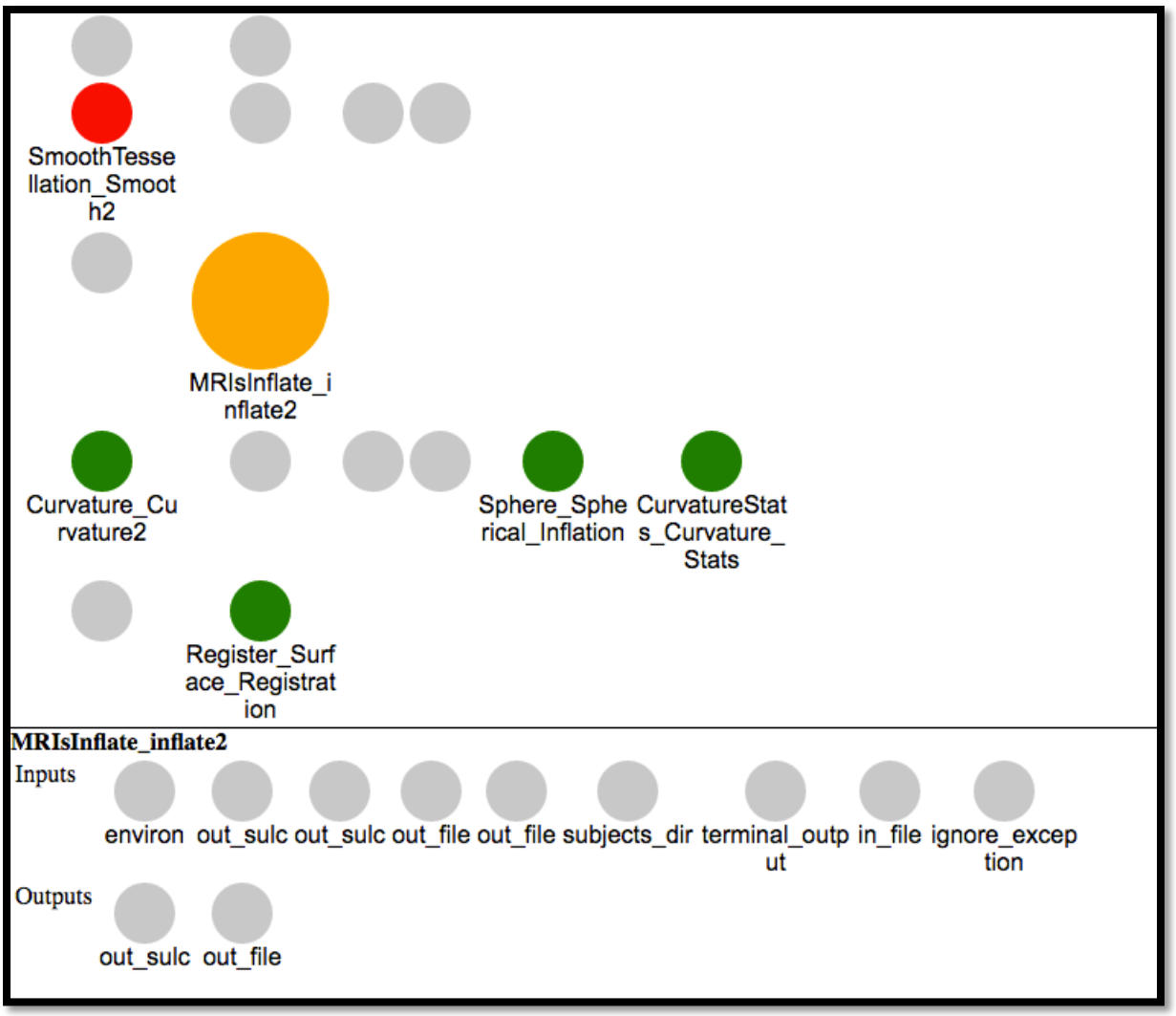

Figure 3: The inputs and outputs of a node are displayed in an information panel on the bottom of the screen.

The information panel adds a new tab for every node that is clicked. These tabs, in addition to the inputs and outputs, contain links that can point to additional information about the node. At this stage in the project, however, these links have no function.

Implementation Details

Workflow Viewer is divided into three main steps in its implementation. The first step converts a TriG file, describing the workflow provenance graph, into an intermediate JSON object. The second step takes in this JSON object and performs further analysis on it to allow for fast querying of necessary information. Finally, the third step displays the graph in the browser.

Step 1: Converting from TriG to JSON

Nipype produces a TriG file that describes the workflow output when it executes. The purpose of the first step in Workflow Viewer is to convert this TriG file into a JSON object that can be readily manipulated in JavaScript directly in the web browser. This conversion is done in Python in the TrigParser.py file. Given a TriG file, TrigParser uses the RDFLib Python module to run SPARQL queries on the data and uses the results of these queries to build a python object. This python object's structure exactly matches the structure of the necessary JSON object required in step 2, so that when this object's construction is complete, TrigParser simply dumps this object into a JavaScript file as a JSON object.

Because TriG conversion takes place in Python, this step cannot be run in the user's web browser. Instead, the user must upload a TriG file to the server, where the server then runs TrigParser.py on the file and returns the required JSON object. As a potential solution to this problem, so that all of Workflow Viewer can be run directly in the browser, there exist JavaScript libraries, such as rdflib.js, that allow for the execution of SPARQL queries in JavaScript. At this point in time, Workflow Viewer does not use these libraries.

Step 2: Augmenting the JSON Object

Once the JSON object has been created, the rest of the graph processing takes place in JavaScript and therefore runs directly in the browser. The JSON object is passed into a JavaScript module called workflow_viewer_graph that follows the JavaScript module pattern[iii]. This module runs additional analysis on the JSON object and then returns a collection of getter methods that allow for fast querying of information that users may need.

When the workflow_viewer_graph module first takes in the JSON object, it makes a deep copy of the object and throws away the reference to the original object. Thus, any changes it makes to the object internally are not reflected in the original JSON object. This is done with the purpose of modularity in mind, so that the workflow_viewer_graph module is completely independent of any other processes running on the browser, including ones that may also be using the original JSON object.

Once the copy has been made, workflow_viewer_graph hashes all of the ids of the nodes, inputs, and outputs. This step is necessary because html elements that represent these graph objects (in step 3) will have the same ids as these objects. Because html (and JQuery) standards for id naming are stricter than RDF standards, workflow_viewer_graph needs a way to convert its given ids into new ids that conform to these standards. This is accomplished with hashing.

Next, workflow_viewer_graph augments its internal JSON graph representation by adding a “children” array to each node object. Because the original JSON object only contains a “parents” array for each node, this step is necessary to allow for constant-time querying of a node’s children.

Finally, workflow_viewer_graph builds a topological table of the graph. This table is a double-array, where rows contain nodes that are topologically

equivalent (i.e. the longest path from a start node to any node in the row is of the same length). The ordering of the rows is topological in increasing order, so that start nodes occupy row 0 (the first index), and end nodes with the longest paths from start nodes occupy the final row. Because Nipype workflow graphs are always acyclic and directed, they can always be topologically sorted into such a table.

When these augmentations are completed, workflow_viewer_graph returns a collection of getter methods that allow for rapid accessing of useful information about the graph. These methods are listed below.

```
{
        get_topological_table()
        get_parents(node_id)
        get_children(node_id)
        get_label(node_id)
        get_inputs(node_id)
        get_outputs(node_id)
        get_child_with_input(node_id, output_id)
}
```

Figure 4: A collection of methods that workflow_viewer_graph returns.

Each of these methods either returns an immutable object or a deep copy of mutable objects so as to maintain modularity and defend its internal representation from outside changes.

Step 3: Displaying the Graph in the Browser

The final step is to display the graph in the browser. This is done in the workflow_viewer_display module, written in JavaScript. This module is responsible for building the html elements that are to be displayed and adding click/hover functionality to those elements. The JQuery (1.9.0) library[iv] is used heavily throughout this module to facilitate these tasks. This module is also responsible for populating the information panel with node information whenever a node is clicked in the visual graph. The Workflow Viewer project also comes with an index.html file where the html is displayed, and a workflow_viewer_style.css file that provides the necessary styling needed to display the html correctly.

Putting it all Together

The Workflow Viewer application comprises all three steps, allowing a user to visually display and investigate a topologically sorted workflow graph produced by Nipype. The illustration below shows the interfacing between all three steps.

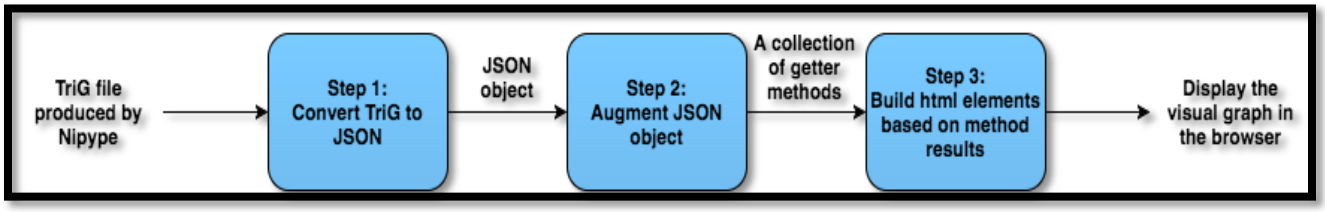

Figure 5: A visual representation of the data processing in Workflow Viewer.

Specifications

Since Workflow Viewer uses three main modules to perform its data processing, the inputs to each of these modules must conform to a specification to ensure that the module works properly. These specifications are given below.

Spec for Input to Step 1:

The module in step 1 takes in a TriG file that is produced by a Nipype execution. This TriG file must describe a Nipype workflow (that is therefore directed and acyclic) and must conform to the W3C PROV standard.

Spec for Input to Step 2:

The JSON object that the workflow_viewer_graph module takes in must be structured as below:

```
{
  "node_id1": {
                "label": "node_label",
                "parents": ["parent_node_id1", ...],
                "info_collections": **See Figure 7**
  },
  "node_id2": {
                ...
  },
  ...
}
```

Figure 6: The spec for the JSON object that workflow_viewer_graph takes in.

```
"info_collections": {
                        "inputs": [
                                    {
                                      "node_id": "node_id",
                                      "input_id": "input_id",
                                      "name": "input_name",
                                      "value": "input_value"
                                    },
                                    ...
                        ],

                        "outputs": {
                                     "out_key1": {
                                                   "name": "output_name",
                                                   "value": "output_value"
                                     },
                                     ...
                        },
}
```

Figure 7: The spec for an "info_collections" array that goes inside a node object in Figure 6.

The JSON object may contain more fields, but at the very minimum, it must contain the fields listed above. Each node object has an info_collections array that contains various pieces of information about the node. The inputs array goes inside the info_collections array. Each input object, in addition to containing an input_id, must also contain the node_id of the node that the input came from

(i.e. the node_id for which it is an output). Of course, this JSON object must also describe an acyclic graph.

Spec for Input to Step 3:

The input to the workflow_viewer_display module in step 3 is a JavaScript variable named workflow_viewer_graph. This variable must contain the following functions.

```
workflow_viewer_graph {

        /*
         * returns a topologically sorted double-array
         * of node_ids
         */
        get_topological_table()

        /*
         * returns an array of all node_ids that are
         * parents of the node with node_id
         */
        get_parents(node_id)

        /*
         * returns an array of all node_ids that are
         * children of the node with node_id
         */
        get_children(node_id)
```

```
        /*
         * returns the label of the node with node_id
         */
        get_label(node_id)

        /*
         * returns an array of all input objects that
         * belong to the node with node_id
         */
        get_inputs(node_id)

        /*
         * returns an object of all output objects
         * that belong to the node with node_id
         */
        get_outputs(node_id)

        /*
         * returns an array of node_ids that have
         * an input with id output_id and come from
         * the node with id node_id
         */
        get_child_with_input(node_id, output_id)
}
```

These are all the functions that workflow_viewer_display needs to display the graph and add the desired functionality. It is also implied that each object returned by these methods is either immutable or is a deep copy, so that they may be mutated freely if necessary.

Future Work

There remains work that must be done to improve Workflow Viewer. One unfinished task is to make Workflow Viewer run completely in the browser. Because step 1 relies on a Python file, the first step cannot be run directly in the browser. To overcome this, workflow viewer will have to incorporate a JavaScript library, such as rdflib.js[v], that supports parsing of TriG files and running SPARQL queries on the data.

Another task is to make Workflow Viewer run in older browsers. During its construction, Workflow Viewer was only tested in Google Chrome (Version 50.0.2661.94) and Safari (Version 7.0.1). It is guaranteed to work in these browsers with version numbers greater than or equal to these versions, but not in any other browser. Further testing and tweaking is required to make the application compatible across many more browsers.

Finally, a major task is to find and implement a better way to fit the graph in the browser. Showing a topologically sorted table of nodes is a good start, but it can lead to rows that are proportionally longer than others. This single row may double or triple the width of the web page, making it necessary to scroll every time a user wishes to investigate the graph. To solve this, Workflow Viewer can incorporate a “cover flow” functionality across rows, so that nodes are compressed together but unfold if the mouse hovers over them.

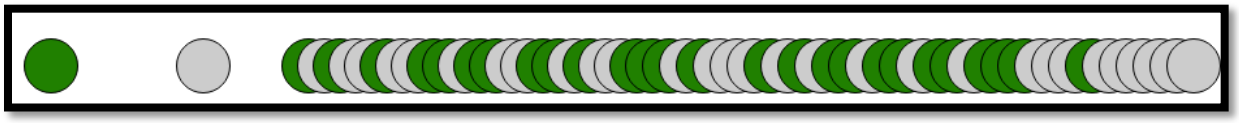

Figure 8: A possible way to compress long rows into the page width, using a cover flow functionality.

Acknowledgements

I would like to thank my advisor, Satrajit Ghosh, for all of his help and excellent guidance throughout the course of this project. I would also like to thank him for giving me the opportunity to work on this project with him—I learned a lot about Nipype and the semantic web and had a great time working on this project.

References

[i] **Nipype:** http://nipy.org/nipype/

[ii] **W3C PROV Standard:** https://www.w3.org/TR/prov-overview/

[iii] **JavaScript Module Pattern:** http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html

[iv] **JQuery Library:** https://jquery.com/

[v] **RDFLib.js:** https://github.com/linkeddata/rdflib.js/