Nipype Beginner's Guide

for Nipype v0.4

Release 0.5

Michael Notter

August 18, 2011

CONTENTS

CHAPTER

ONE

WELCOME TO NIPYPE!

1.1 But what is Nipype?

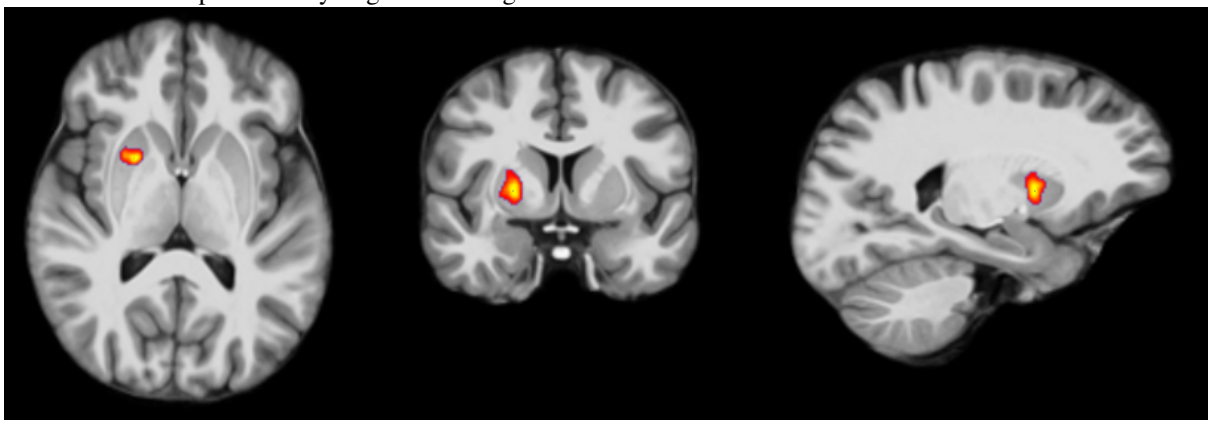
Nipype (Neuroimaging in Python Pipeline: Pipelines and Interfaces) is an open-source, user-friendly, community-developed, Python-based software package that provides an uniform interface to existing neuroimaging softwares like:

AFNI, BRAINS, Camino, Camino-TrackVis, ConnecomeViewerToolkit, dcm2nii, Diffusion Toolkit, FreeSurfer, FSL, Nipy, NiTime, Slicer, SPM, SQLite, PyXNAT

...and facilitates the interaction between these packages by using workflows. It is able to let **shell** (FSL, AFNI, Camino), **MATLAB** (SPM) and **Python** (Nipy) scripts communicate between each other and provides an environment that encourages interactive exploration of algorithms, eases the design of workflows within and between packages and reduces the learning curve necessary to use this different packages. For more informations about Nipype go to the nipype homepage.

1.2 What can you expect from this user guide?

The goal of this beginner's guide is simple. With its help you should be able to take the raw data from your scanner and execute all steps necessary to get something like this:



This means that you are able to prepare your data (e.g. recon-all, dicom2nifti conversion), run a full first and second level analysis (on the volume and surface), create your own normbrain template (have a nice normbrain, that is closer to your actual data), create pictures of brain activity like this and extract the statistical relevant values (extraction of anatomical and functional regions of interests (ROIs)). All that by using Nipype. But that's not all...

This user guide will also teach you all the basics of Nipype that you need to know. How a pipeline is structured and how to build one by yourself. It will show you different ways to visualize your pipelines. Not yet included is the chapter about ANTS which will teach you how to create your own normbrain template and how to normalize your data from subject space into a common template space.

Note that this guide is meant as a general introduction. The implementation of Nipype is nearly unlimited and there is a lot of advanced knowledge that won't be covered by this guide but can be looked up at various places on the nipype homepage.

There are also a lot of very good tutorials specific for different usages of Nipype like using FreeSurfer for smoothing, using FSL for DTI analysis, for fMRI analysis with FEEDS data, for fMRI analysis or using SPM for analysis on auditory dataset etc. They all can be found in the tutorial section of the Nipype homepage.

Important: It is important to mention that the examples of this beginner's guide are tuned for the environment of the Gabrieli Lab at MIT. So it might be possible that you have to change some paths or environment variables.

1.3 What if you're lost?

If you have any questions about Nipype and can't find the answer on the nipype homepage feel free to contact the Nipype mailing list or to search its archive. Sign up for the mailing list here.

If you have any questions about this Beginner's Guide or want to give its author any kind of feedback or suggestions, please contact me at mnotter@mit.edu. It's highly appreciated.

1.4 Before we can start... Preparation!

Make sure that you have installed all the necessary modules and software on your system. For more information go to the download and install section.

Before you can run a nipype python script make also sure to set up the necessary environment variables, meaning to source to your corresponding Nipype and FreeSurfer and to export the corresponding FreeSurfer and Matlab paths. With the current Nipype v0.4 and FreeSurfer v5.1.0 the terminal commands can look like that:

```
source /software/python/setup-nipype-0.4.sh
source /software/Freesurfer/5.1.0/SetUpFreeSurfer.sh

export MATLABCMD=$pathtomatlabdir/bin/$platform/MATLAB

export FREESURFER_HOME=/software/Freesurfer/5.1.0
export FSFAST_HOME=/software/Freesurfer/5.1.0/fsfast
export SUBJECTS_DIR=/software/Freesurfer/5.1.0/subjects
export MNI_DIR=/software/Freesurfer/5.1.0/mni
```

After all installations are done and you've set up all necessary environment variables you can start iPython:

```
ipython
```

Now you're good to go!

CHAPTER

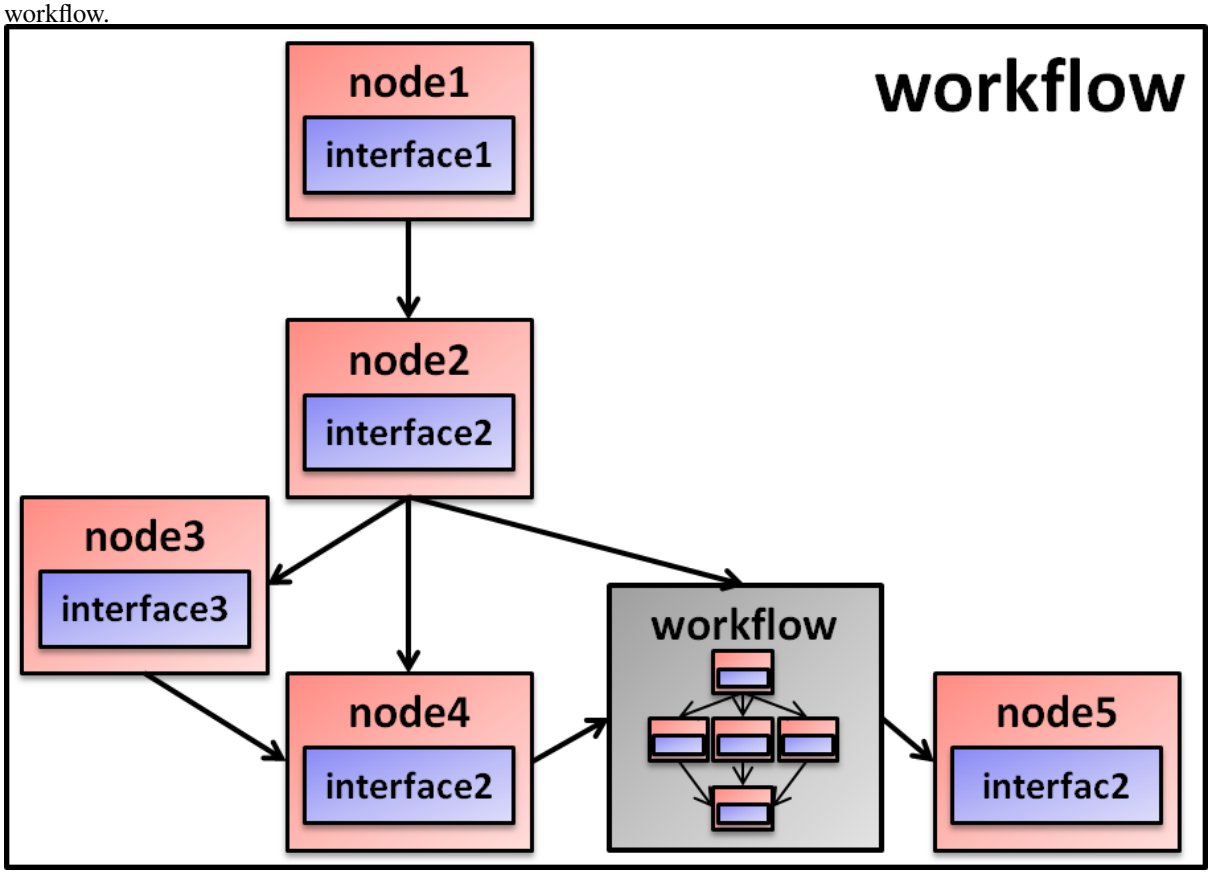
TWO

HOW TO BUILD A PIPELINE

This section is meant as a step by step introduction to building your own pipeline. At the end you should know what the important characteristics of a pipeline is, how it is constructed, how its parts are connected, so that you are ready to implement your own pipeline.

2.1 What is a pipeline?

A pipeline in the Nipype sense is a sequence of procedures to automate the analysis of fMRI-data. A pipeline is also called a workflow and is built by connecting specific nodes to each other. In the context of nipype, nodes contain specific functions or algorithms of interfaces such as SPM, FSL, FreeSurfer etc. In order for interfaces to be used in a workflow they need to be encapsulated into a node. All those nodes have defined inputs and outputs. Creating a workflow then is a matter of connecting appropriate outputs to inputs. The great thing about Nipype is that those workflows themselves than can act as a node inside another workflow leading to a complex data analysis workflow.



Nipype explicitly implements a pipeline as a graph. This makes it easy to follow what steps are being executed and in what order. It also makes it easier to go back and change things by simply reconnecting different outputs

and inputs or by inserting new nodes.

2.2 How to build your own pipeline

There are many ways to construct a pipeline but in the end it comes down to the following steps:

1. **Import appropriate modules**
2. **Define nodes**
3. **Define pipeline(s)**
4. **Create connections**

2.2.1 Import appropriate modules

The first thing you'll have to do is to import the interfaces you want to use. That depends on the nodes and algorithms you want to use in your pipeline. You can either import an interface and give it a specific name or import only a desired algorithm of it.

```
# imports the engine interface as 'pe'
import nipype.pipeline.engine as pe

# imports only the function Bunch from the base interface
from nipype.interfaces.base import Bunch
```

Important: If you want to use the output of FreeSurfer's recon process, please make sure to tell FreeSurfer where the subjects directory is by using the following command:

```
import nipype.interfaces.freesurfer as fs
freesurfer_dir = '~SOMEPATH/freesurfer_data'
fs.FSCommand.set_default_subjects_dir(freesurfer_dir)
```

2.2.2 Define nodes

Node Initiation

Before the parameters of a node can be specified they first have to be initiated. The initiation is quite simple and done as follows:

```
nodename = pe.Node(interface=interface.algorithm(), name='visibleName')
```

- **nodename**: name of the variable which identifies the node in the code
- **pe.Node**: defines the characteristic of the node, which can be a Node, a MapNode or a Workflow
- **interface**: name of the imported interface you want to use (e.g. SPM, FSL, FreeSurfer,...)
- **algorithm**: name of the algorithm you want the node to execute
- **visibleName**: name which is used for the naming of the folder the node output is stored in and the name of a node in the pipeline graph (recommended to be the same as nodename)

```
#Example of an initiation of a spm-realignment node
import nipype.interfaces.spm as spm
realign = pe.Node(interface=spm.Realign(), name='realign')
```

Hint: The difference between a Node and a MapNode is explained in more detailed here. But briefly explained is a MapNode a sub-class of Node that allows interfaces that normally operate on a single input to execute the interface on multiple inputs.

Node Parameters

Depending on the purpose of a node and its underlying algorithm, different parameters can be specified. They can be distinguished into:

1. **mandatory inputs**: inputs that have to be given
2. **optional inputs**: inputs to get the node to behave in a specific way
3. **outputs**: the possible outputs that a node creates

But how can you find out what the possible inputs and outputs of a node are?

- check the section Interfaces and Algorithms on the nipype homepage by clicking on the node you're interested in
- use the `help()` method of a module in iPython (e.g. `fsl.Smooth.help()`)
- view the docstring of a module in iPython (e.g. `fsl.MCFLIRT?`) which shows you the documentation and an example of an implementation in the command window.

Note: Be aware that some algorithms' input options should not be set together (mutual exclusion) while other inputs need to be set as a group (mutual inclusion).

Node Specification

The specification of a node can be done in three ways.

```
import nipype.interfaces.fsl

#1. specify parameters in the during initiation
mybet = fsl.BET(in_file='foo.nii', out_file='bar.nii')

#2. specify parameters after initiation
mybet = fsl.BET()
mybet.inputs.in_file = 'foo.nii'
mybet.inputs.out_file = 'bar.nii'

#3. specify parameters when running a node
mybet = fsl.BET()
mybet.run(in_file='foo.nii', out_file='bar.nii')
```

Iterables

If you want a node to be executed over different sets of data (e.g. different subjects, different conditions, different smoothing kernels,...) you have to use iterables. Iterables are a very easy way to explore the impact of variations of your data analysis workflow.

```
nodename.iterables = ('input_to_iterate_over',[conditions_to_iterate_over])
```

E.g. If you want to execute a pipeline for subject1 and subject2 and you want to run the pipeline with a smoothing kernel of 4 and one of 8 you do the following:

```
startnode.iterables = ('subject_id', ['subject1','subject2'])
smoothnode.iterables = ('fwhm', [4, 8])
```

Note: Iterables splits the following workflow into subworkflows and creates an indexed copy not only of the node but also of all the nodes dependent on it.

Iterfields

If you'll use MapNodes you'll also have to use an iterfield. This enables running the underlying interface over a set of inputs and is particularly useful when the interface can only operate on a single input. A good tutorial to iterables and iterfields can be found at MapNode, iterfield, and iterables explained

Stand-alone node

If you want to run an algorithm without being a node or being a part of a pipeline you only have to define the nodename, interface and algorithm.

```
smooth = spm.Smooth()
```

This is most of the time used if you want to test a node or if you want to use the nipype environment to run the different kind of algorithms without using them inside a pipeline. For example, it isn't recommended to use the `recon-all` algorithm inside a pipeline or more extreme in parallel mode because of its computational and time costs. Nonetheless you can use Nipype to execute the recon-all process.

Individual nodes

If you want to create an individual node by yourself that doesn't use an algorithm of an interface already specified and you want to use the advantage of input and output fields you can build your own node with the `IdentityInterface` method of the `utility` interface.

```
#import the utility interface
import nipype.interfaces.utility as util

#define the fields you want to use
individualnode = pe.Node(interface=util.IdentityInterface(fields=['field1','field2']),
                         name='individualnodename')
```

Now we have designed our own node with the in- and output field 'field1' and 'field2'. If you now want the node to execute a specific kind of algorithm, you'll have to add a function to the connections to the inputnode. How this can be done is described in **4. Create connections - Modifying inputs to nodes**.

2.2.3 Define pipeline(s)

The initiation of a pipeline is quite the same as the one of a node. Except that you don't need to declare an interface.

```
workflow = pe.Workflow(name='preproc')
```

Cloning

If you've already created a pipeline with all its connections and want to reuse it in another part of the workflow you can simply clone it with the `clone` method.

For example, if you've already created an analysis pipeline for the workflow after a volume preprocess and want now to reuse this algorithm for the analysis of preprocessed surfacedata you can clone the volanalysis like this:

```
surfanalysis = volanalysis.clone(name='surfanalysis')
```

This cloning has to be done, because if you would use the volanalysis again, the data of the volanalysis would be overwritten. Because of that and because of the ambiguity of the connections in the workflow, the pipeline would not run. To solve this problem, every node and pipeline has to have its unique name.

If you want to change some parameters of the pipeline after cloning you just have to specify the exact pipeline, node and parameter you want to change:

```
surfanalysis.inputs.level1design.timing_units = 'secs'
```

Which now would set the input field `timing_units` of the level1design node which is part of the surfanalysis pipeline to 'secs'.

2.2.4 Create connections

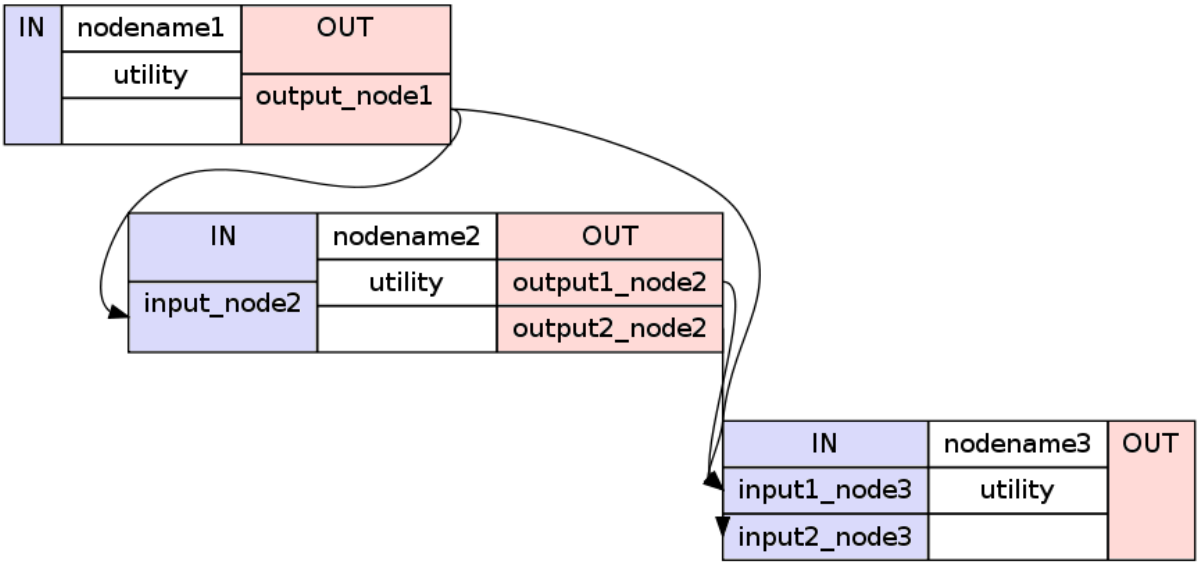
This is the essential part of creating a pipeline and leads to the main advantage of the pipeline which is to execute everything autonomous, in one workflow and if you want to in parallel.

Connect nodes to each other

There is a basic and an advanced way to create connections between two nodes. The basic way allows only to connect two nodes at a time whereas the advanced one can establish multiple connections at once.

```
#basic way to connect two nodes
workflowname.connect(nodename1, 'out_files_node1', nodename2, 'in_files_node2')

#advanced way to connect multiple nodes
workflowname.connect([(nodename1, nodename2, [('output_node1', 'input_node2')]),
                      (nodename1, nodename3, [('output_node1', 'input1_node3')]),
                      (nodename2, nodename3, [('output1_node2', 'input1_node3'),
                                              ('output2_node2', 'input2_node3')
                                              ])
                      ])
```

The advanced connection example would as a detailed graph look like this:



It is important to point out that you don't just have to connect the nodes, but rather to connect the output and input fields of each node.

Connect pipelines to each other (necessary if you have multiple pipelines)

If you have multiple pipelines, like one for preprocessing, one for model estimation and one for volume analysis, you can't just connect the nodes to each other. You have to connect the pipelines to each other instead.

Assumed that we have a node `realign` which is part of a pipeline called `preprocess` and that we have a node called `modelspec` which is part of a pipeline called `modelestimation`. If we now want to connect does two pipelines at those particular points we first have to create a kind of meta-pipeline which contains those to pipelines. This initiation and the following connections would look like that:

```
frameflow = pe.Workflow(name='frameflow')
frameflow.connect([(preprocess, modelestimation,[('realign.out_files', 'modelspec.in_files')
                                                 ])
                    ])
```

You see that the main difference to the connections between nodes is that you connect the pipelines, but have to specify which nodes with which output should be connected to which nodes with which input.

Add node(s) to pipeline (optional)

If you want to run a node by itself without connecting it to any other node, you can do that with the add_nodes method.

```
#adds node smooth and node realign to the pipeline
workflow.add_nodes([smoother, realign])
```

Modifying inputs to nodes

If you want to modify the output of a node before sending it to the next one you can do that by adding a function into the connection process.

First you have to define your function that modifies the data and returns the new output. If you have done this, than you can insert the function into the connection process.

```
#your function that does something
def myfunction(input_from_node):

    #changes the data as you defined
    output_for_node_2 = input_from_node * 2

    #return the output
    return output_for_node_2


#connection of two nodes with a function in between
workflowname.connect([(nodename1, nodename2,[(('out_file_node1', myfunction),
                                              'in_file_node2')]),
                      ])
```

This will take the output of 'out_file_node1' and give it as an argument to the function `myfunction`. The return value that will be returned by `myfunction` then will be forwarded as input to 'in_file_node2'.

If you want to insert more than one parameter into the function do as follows:

```
def myfunction(input_from_node, additional_input):

    output_for_node_2 = input_from_node + additional_input
    return output_for_node_2


#connection of two nodes with a function in between which takes two arguments
workflowname.connect([(nodename1,nodename2,[(('out_file_node1',myfunction,additional_input),
                                             'in_file_node2')]),
                      ])
```

2.3 How to get data into and out of your pipeline

After constructing the framework of the pipeline in the previous section, we're almost ready to execute the pipeline. But first we have to define the data we want to run the pipeline on and the results we want to get out of it. To do

that we have to consider the following points:

1. **Infosource:** to define the list of subjects on which the pipeline will be executed
2. **Datasource:** to grab the data you want to use and insert it into the pipeline
3. **Inputnode:** to distribute the data and experiment specific parameters to the pipeline (optional)
4. **Datasink:** to store specific outputs at a given place (optional but recommended)
5. **Model Specification:** to feed the pipeline with model specific components like contrast, conditions, onset times etc.
6. **Connect new nodes to your pipeline:** to connect the infosource, datasource, inputnode and datasink with the framework of your pipeline

2.3.1 Infosource

The best way to tell a pipeline on which subjects it should be executed on is to build an infosource node. The only thing that this node contains is a list of the subjects and the instructions to execute the pipeline on each of this subjects. This is done with the iterables method.

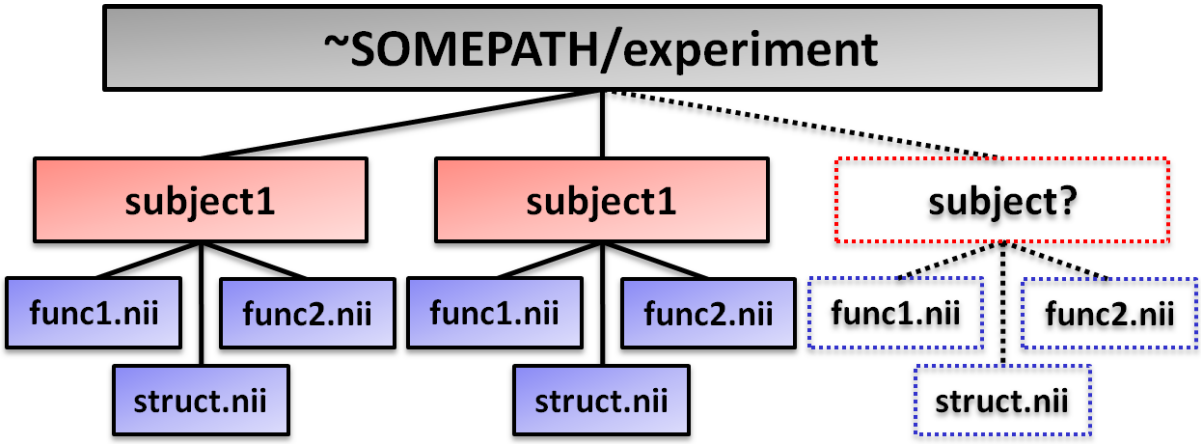
```
#import the utility interface
import nipype.interfaces.utility as util

#initiate the infosource node
infosource = pe.Node(interface=util.IdentityInterface(fields=['subject_id']),
                     name="infosource")

#define the list of subjects your pipeline should be executed on
infosource.iterables = ('subject_id', ['subject1','subject2','subject3'])
```

2.3.2 Datasource

Nipype has its own DataGrabber interface that allows us to define flexible search patterns to get the subject specific data into the pipeline. As the name implies DataGrabber grabs the data from a specified folder and stores its location in a specified output field.

```
#import nipype i/o routines
import nipype.interfaces.io as nio

#initiate the DataGrabber node with the infield: 'subject_id'
#and the outfield: 'func' and 'struct'
datasource = pe.Node(interface=nio.DataGrabber(infields=['subject_id'],
                                               outfields=['func', 'struct']),
                     name = 'datasource')
```

To use the DataGrabber it is important to know what the exact structure of your folders is and where the data is stored at. In this example we assume that the layout of our data is as following:



As you can see all the necessary data is stored in the experiment folder. The data of each subject is stored in its individual subject_folder. The name of this folder changes with each subject. There are two ways how you can define the structure of data you want to grab.

```
#to specify the location of the experiment folder
datasource.inputs.base_directory = '~/experiment_folder'

#define the structure of the data folders and files.
#Each '%s' will later be filled by a template argument.
datasource.inputs.template = '%s/%s.nii'

#First way: define the arguments for the template '%s/%s.nii' for each field individual
datasource.inputs.template_args['func'] = [['subject_id', ['func1','func2']]]
datasource.inputs.template_args['struct'] = [['subject_id','struct']]

#Second way: store all the arguments for the template in a dictionary and ...
info = dict(func=[['subject_id', ['func1','func2']]],
            struct=[['subject_id','struct']])
#... pass it to template_args.
datasource.inputs.template_args = info
```

Note: The values defined in `template_args` will be filled into the placeholders of `template`. Because 'subject_id' is defined as ['subject1','subject2','subject3'] (see definition of infosource node), the outfield 'func' of the datagrabber node will store 'subject1/func1.nii', 'subject1/func2.nii' and 'subject1/struct.nii' in the 'struct' outfield for subject1.

2.3.3 Inputnode

If you want to keep a clearly arranged distribution of the input data it is suggested to create an inputnode that serves that purpose. This inputnode specifies and collects all the inputs that are needed for the workflow and distributes them to specific places in the pipeline.

```
#import the utility interface
import nipype.interfaces.utility as util

#define the inputnode with the fields you want to distribute
inputnode = pe.Node(interface=util.IdentityInterface(fields=['func',
                                                             'subject_id',
                                                             'session_info',
                                                             'contrasts']),
                    name='inputnode')
```

2.3.4 Datasink

Sometimes you want to store selected output at a specified and easy accessible place so that you don't have to search for it in the depth of your working directory, where all in- and outputs of each and every node are stored at. For this purpose the DataSink interface was created:

```
#import nipype i/o routines
import nipype.interfaces.io as nio

#initiate node
datasink = pe.Node(interface=nio.DataSink(), name="datasink")

#specify the name and location of the datasink folder
datasink.inputs.container = 'name_of_datasink_folder'
datasink.inputs.base_directory = '~/experiment_folder'

#define the outputs you want to store by connecting them to the datasink node
metaflow.connect([(frameflow,datasink,[('preproc.bbregister.out_reg_file',
                                        'bbregister'),
                                       ('volanalysis.contrastestimate.spm_mat_file',
                                        'spm_mat_file'),
                                       ])
                  ])
```

Note: The name that you give the input-filed of the DataSink node (here 'bbregister' and 'spm_mat_file') will be taken as a name giver for the subfolder where those specific files will be stored in the DataSink folder.

The DataSink node is really useful to keep control over your storage capacity. If you store all important files that you'll need for further analysis in this folder you can delete the `workingdir` of the pipeline after executing and counteract storage shortage. You can even set up the configuration of the pipeline so that it will not create a working directory at all. For more information go to Configuration File.

Hint: For a more detailed explanation to datasource and datasink go to: DataGrabber and DataSink explained.

2.3.5 Model Specification

Before we can run our pipeline we have to feed it with model specific components. For a first level pipeline this would be the names of the conditions, the contrasts and onset times. In this part we might also want to add some parametric modulators, regressors, etc.

Contrasts

To insert all the contrast specific values into the pipeline we first have to save them into a variable, in this case called `contrasts`. The structure of this variable is a list of lists. The inner list specifies the contrasts and has the following format - [Name, Stat, [list of condition names], [weights on those conditions]. The condition names must be the same we later feed into `subjectinfo` function described below.

```
#Names of different conditions
namesOfConditions = ['basic','condition1','condition2','condition3']

#contrasts for all sessions
contrast_1 = ('basic vs. conditions','T', namesOfConditions, [3,-1,-1,-1])
contrast_2 = ('all vs. condition1',  'T', namesOfConditions, [0,1,0,0])
contrast_3 = ('all vs. condition2',  'T', namesOfConditions, [0,0,1,0])
contrast_4 = ('all vs. condition3',  'T', namesOfConditions, [0,0,0,1])

#contrasts for e.g. session 1 and 3 out of ['session1','session2','session3']
contrast_5 = ('1+3 vs. condition1',  'T', namesOfConditions, [0,1,0,0],[1,0,1])
contrast_6 = ('1+3 vs. condition2',  'T', namesOfConditions, [0,0,1,0],[1,0,1])
contrast_7 = ('1+3 vs. condition3',  'T', namesOfConditions, [0,0,0,1],[1,0,1])

```

```
#store all contrasts into a list
contrasts = [contrast_1,contrast_2,contrast_3,contrast_4,
             contrast_5,contrast_6,contrast_7,contrast_8]

#feed those contrasts to the inputnode filed 'contrasts'
frameflow.inputs.inputnode.contrasts = contrasts
```

Session info

Here we create a function that returns session specific information about the experimental paradigm. This is needed by the SpecifyModel function to create the information necessary to generate an SPM design matrix. This function `subjectinfo` is used to feed the inputnode `session_info` for each subject with the paradigm conditions.

```
def subjectinfo(subject_id):

    #import Bunch from interface base
    from nipype.interfaces.base import Bunch

    #restate the names of
    namesOfConditions = ['basic','condition1','condition2','condition3']

    #Onset Times in seconds for condition ['basic','condition1','condition2','condition3']
    onsetTimes = [[1,10,42,49.6,66.1,74.1,97.6,113.6,122.2,130.2,137.2,153.7,169.2,
                   185.7,201.8,290.4,313.4,321.4,377.5,401.5,410,418.6,442.1,473.6],
                  [17.5,82.1,89.6,145.2,225.3,242.3,281.4,426.6],
                  [26,162.2,209.3,249.3,265.9,205.4,450.1,386],
                  [34,273.4,329.5,338.5,354,362,370,466.4]
                  ]

    #to define two parametric modulators for 'condition1','condition2','condition3'
    para_modu = [None,
                 base.Bunch(name=['target2','target3'], poly=[[1],[1]],
                            param = [[0,0,1,0,0,0,0,0],[0,0,0,0,1,0,0,1]]),
                 base.Bunch(name=['target2','target3'], poly=[[1],[1]],
                            param = [[0,0,0,1,1,1,0,0],[1,0,0,0,0,0,1,1]]),
                 base.Bunch(name=['target2','target3'], poly=[[1],[1]],
                            param = [[0,1,0,0,0,1,0,1],[0,0,0,0,1,0,1,0]]),
                 ]

    #to feed this information to the inputnode we have to store the information
    #in a list 'output' which we will return later
    output = []

    #the parameters get added three times if we have three sessions like in this example.
    #if you need to, you would be able here to specify the session specific parameters
    #for each session differently
    for r in range(3):
        output.append(Bunch(conditions=namesOfConditions,
                            onsets=onsetTimes,
                            durations=[[8] for s in namesOfConditions],
                            amplitudes=None,
                            tmod=None,
                            pmod=para_modu,
                            regressor_names=None,
                            regressors=None))
    return output
```

Note: A detailed instruction on how to set the model specific parameters can be found in the in the Model Specification section.

2.3.6 Connect new nodes to your pipeline

Before you can run your pipeline you will have to connect infosource, datasource, inputnode and datasink to each other and to the pipelines of your framework workflow, here called frameflow. For this purpose you will have to create a meta pipeline.

```
#initiate the meta workflow
metaflow = pe.Workflow(name='metaflow')

#connect infosource, datasource and inputnode to each other
metaflow.connect([(infosource, datasource,[('subject_id','subject_id')]),
                  (datasource,inputnode,[('func','func'),
                                         (('subject_id', subjectinfo),'session_info'),
                                         ]),
                  #connect the inputnode to your workflow
                  (inputnode, frameflow,[('func','surfsmooth.in_file'),
                                         #...etc...
                                         ])
                  #connect output you want to be stored into datasink
                  (frameflow,datasink,[('preproc.bbregister.out_reg_file',
                                        'bbregister'),
                                       ('volanalysis.contrastestimate.spm_mat_file',
                                        'spm_mat_file'),
                                       ])
                  ])
```

2.4 How to run your pipeline

After all nodes were implemented, all parameters were specified and all connections were established, you are able to run your pipeline. This can be done by calling the `run()` method of the `metaflow`.

```
#If you want to run your pipeline in serial
metaflow.run(plugin='Linear')

#If you want to run your pipeline in parallel using two cores
metaflow.run(plugin='MultiProc', plugin_args={'n_procs' : 2})
```

If the `run()` method is called twice, the workflow input hashing mechanism ensures that none of the nodes are executed during the second run if the inputs remain the same. If, however, some parameters change, some of the nodes would have to rerun. Nipype automatically determines which Nodes require rerunning and keeps the computation time to its minimum.

Hint: More informations about how to run your pipeline in a distributed system can be found here: Distributed processing with nipype.

More informations about how to change the configuration of the `run()` method can be found here. There you can learn how to change the configuration so that every node gets fully executed everytime, how to specify that only the necessary output gets stored into the working directory, etc.

CHAPTER

THREE

HOW TO VISUALIZE A PIPELINE

3.1 What kind of graph do you need?

After the connection of all nodes of a pipeline is established we are able to use the `write_graph()` method of the workflow and create a graph. Such a graph gives us a very good overview of our pipeline. For the example in the previous chapter, the command would be:

```
metaflow.write_graph(graph2use='flat')
```

Nipype can create four different kinds of graphs by setting the variable `graph2use` to the following parameters:

- `orig` shows only the highest workflow
- `flat` shows all workflows with subworkflows
- `exec` shows all workflows with subworkflows and expands iterables into subgraphs
- `hierarchical` shows all workflows with subworkflows but maintains their hierarchy

All versions create two graph files except the hierarchical one. The difference of those two files is in the level of detail they are showing. There is a **simple overview graph** called `graph.dot` that shows you the basic connections between nodes and a **more detailed overview graph** called `graph_detailed.dot` that shows you additionally the outputs and inputs of each node and the connections between them. The **hierarchical graph** on the other side creates only a simple overview graph. But its iterable nodes are marked in a different gray tone and subgraphs are surrounded by a square to show their range.

Note: The graph files can be found in the highest pipeline folder of your working directory.

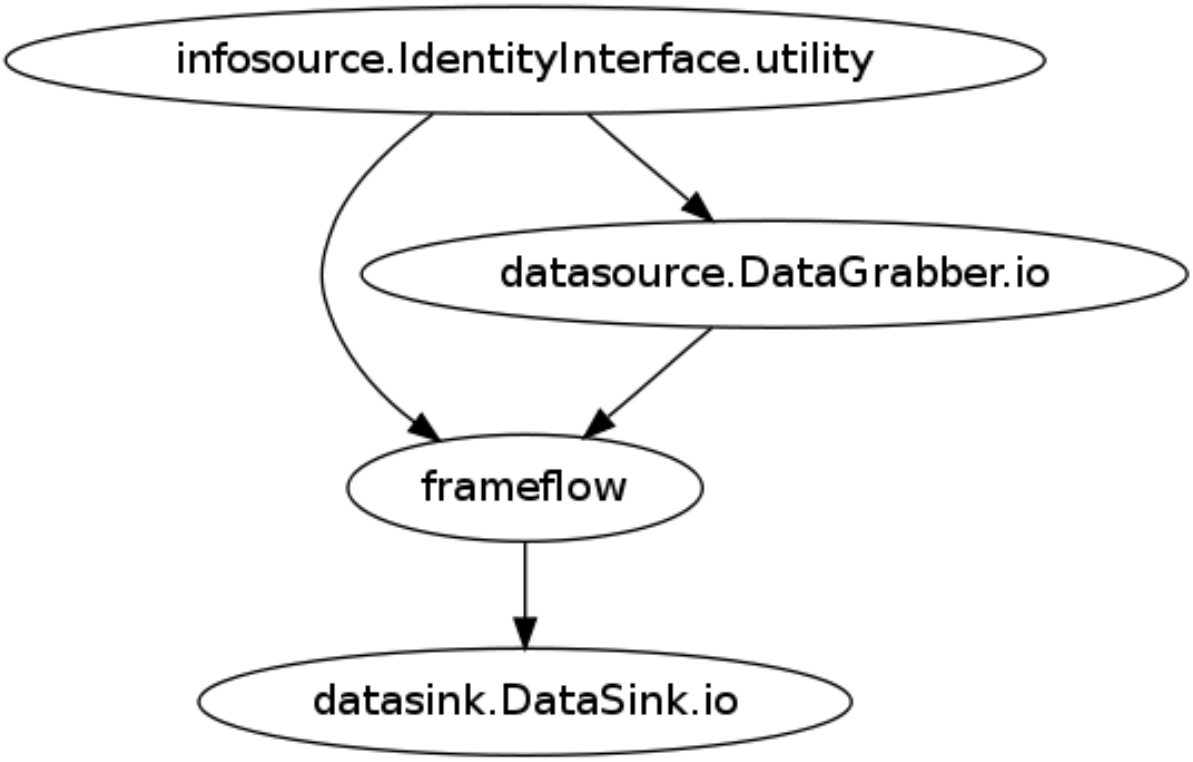
If graphviz is installed the dot files will automatically be converted into png-files. Otherwise you can take the dot files and load them in a graphviz visualizer elsewhere.

3.2 Example graphs

The following graphs are a visualization of the **first level analysis pipeline** or `metaflow` which you will encounter in the chapter: How To Build A First Level Pipeline

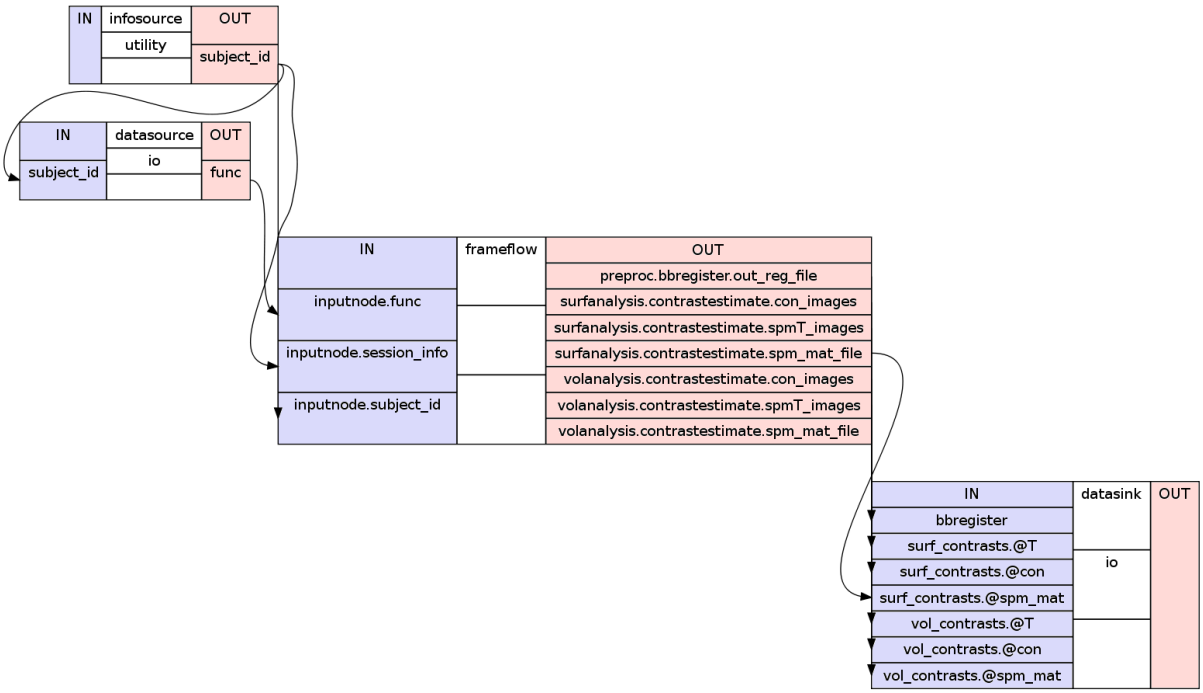
3.2.1 `orig` - simple graph

This is the simple graph of the `orig` version and shows as an example the `metaflow` which you will encounter in a latter chapter. The name of the nodes is composed in the format of `nodename.algorithm.interface`. Because this version of graphs only shows the highest workflow all the subgraphs contained in the `frameflow` are hidden.



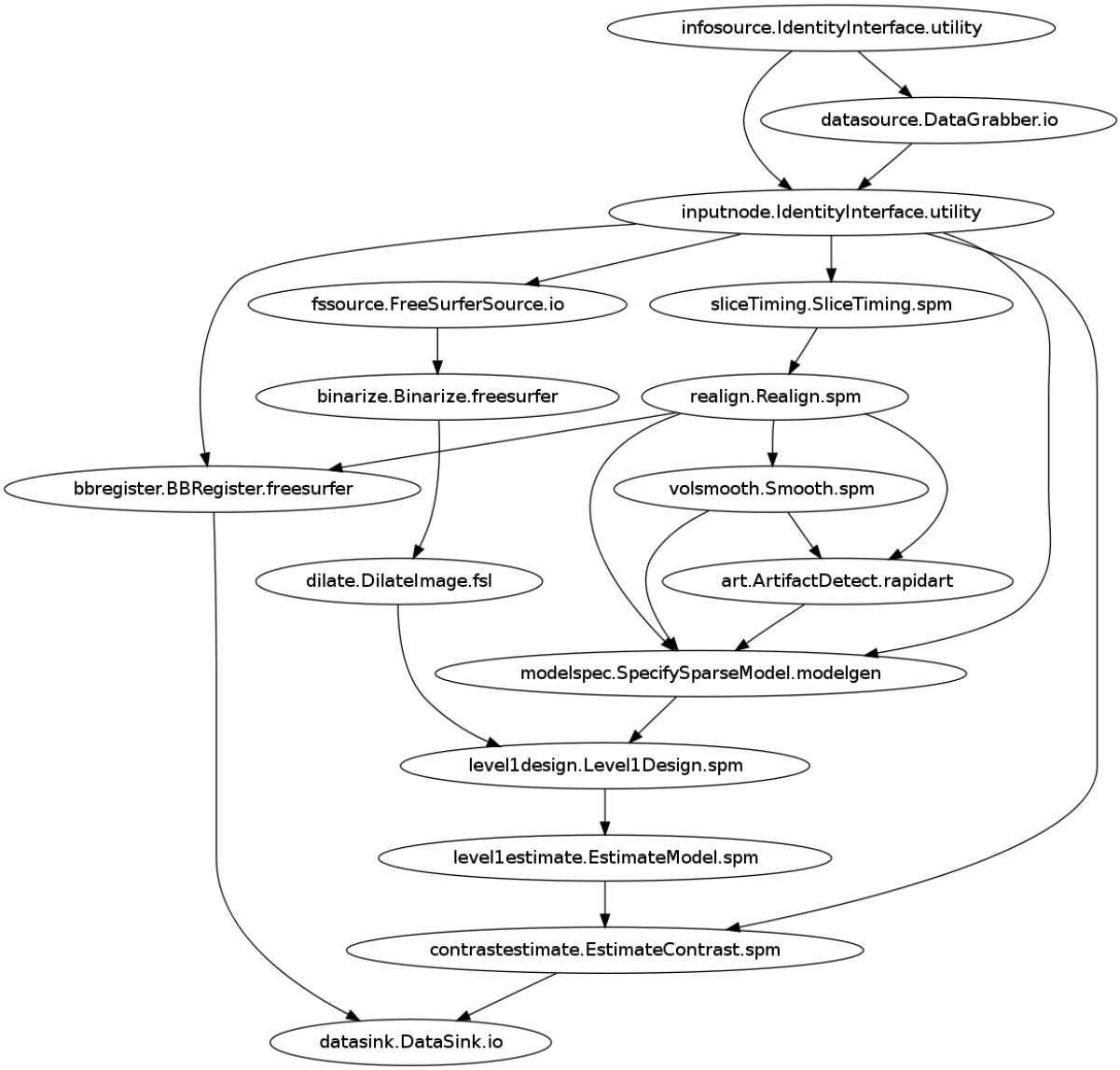
3.2.2 `orig` - detailed graph

This detailed graph of the `orig` version shows the `metaflow` from the simple graph above but with a bit more informations. Now you can see which input of which node is connected to another. Because the subworkflow of the `frameflow` are hidden you only see what inputs and outputs go into this `frameflow` but not more.



3.2.3 `flat` - simple graph

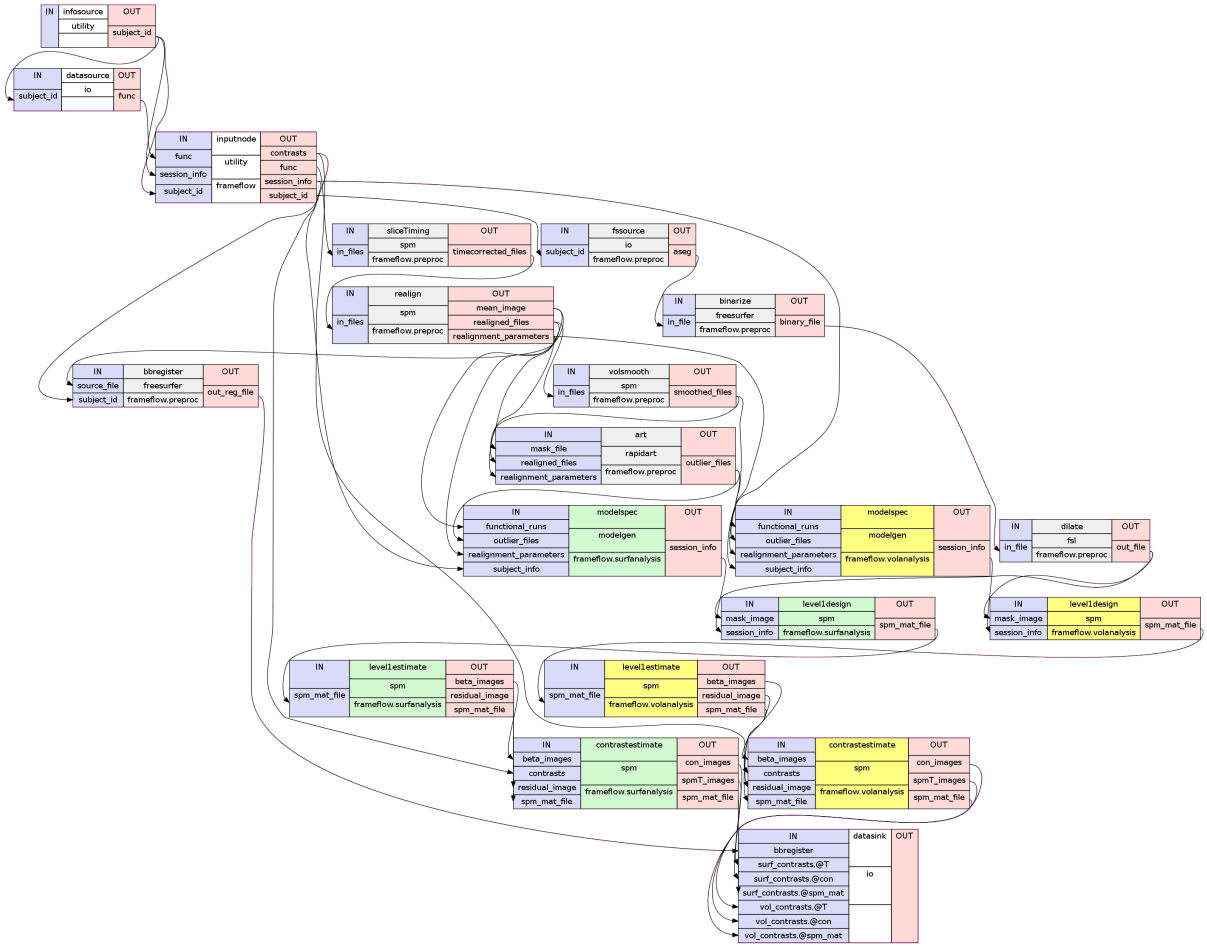
This simple graph of the `flat` version shows the `metaflow`. You can see now that the `frameflow` was expanded by its subnodes.



But this version still doesn't show you if there are different kinds of subworkflows.

3.2.4 `flat` - detailed graph

This detailed graph of the `flat` version shows the `metaflow`. You can see now all nodes, their inputs and outputs and how they are connected to each other.



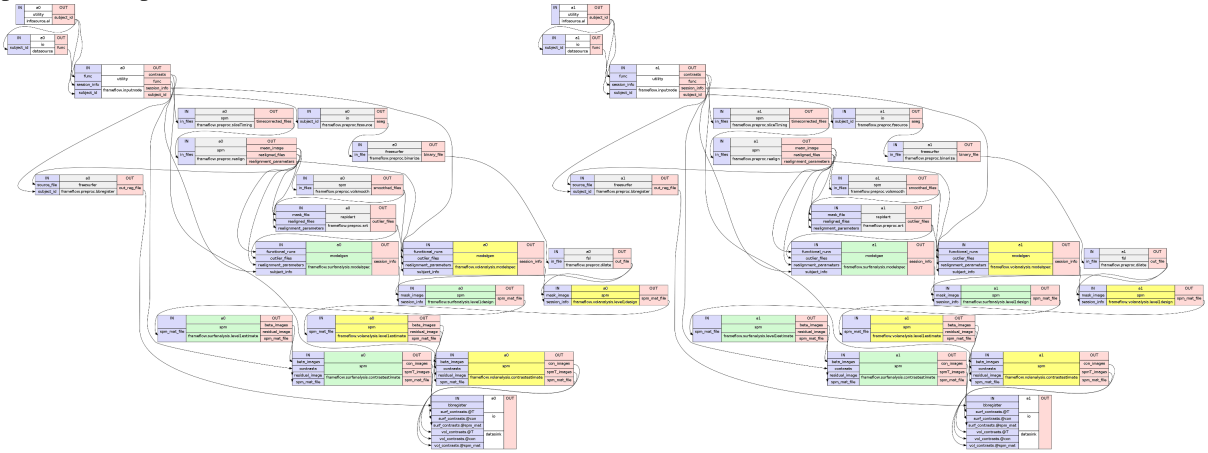
In this version you also see that the that the `frameflow` has actually three subworkflows called `preproc`, `surfanalysis` and `volanalysis`.

3.2.5 `exec` - simple graph

This graph doesn't show you anything different than the simple graph of the `flat` version.

3.2.6 `exec` - detailed graph

This detailed graph of the `exec` version shows the `metaflow`. You can see that you get the same level of detail as if you would run it with `flat` but all iterables are expandend so that you can see all the nodes that would be processed in parallel are shown at once.

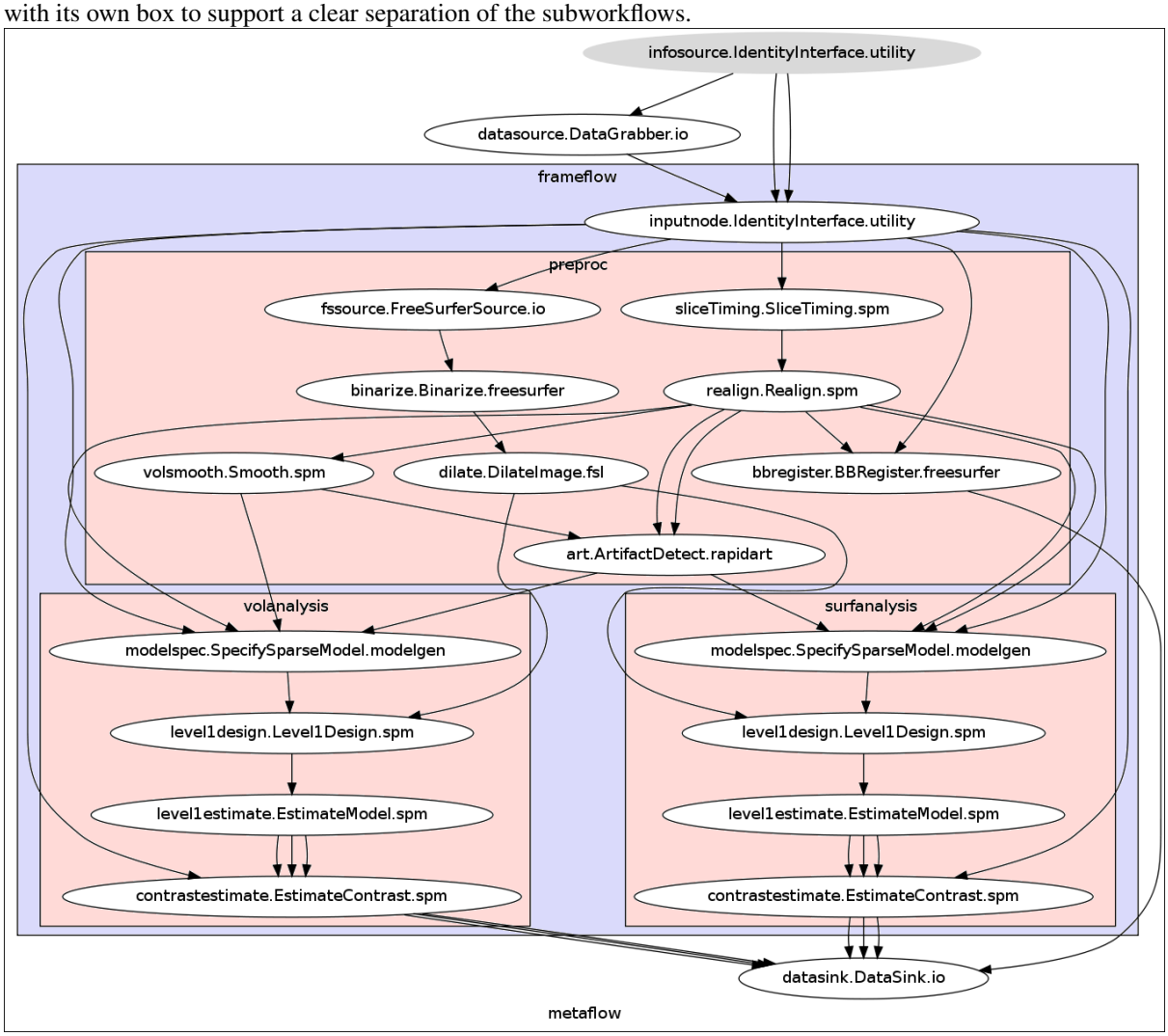


In this case the infosource node iterates over `subject1` and `subject2`.

Note: As you can see from this graph every iteration creates a subgraph with its own index, in this case `a0` and `a1`. Because the iteration of this workflow starts with the infosource node, the indexing already starts there.

3.2.7 `hierarchical` - simple graph

This graph of the `hierarchical` version shows the `metaflow`. This graph doesn't show you the inputs and outputs of each node, but it shows you how many connections exist between nodes and it surrounds each workflow with its own box to support a clear separation of the subworkflows.



In this example you clearly see that the `metaflow` contains a `frameflow` which itself contains three subworkflows `preproc`, `volanalysis` and `surfanalysis`.

Note: The coloration of this graphs was done manually and don't get created by nipype itself.

CHAPTER

FOUR

HOW TO PREPARE YOUR DATA FOR NIPYPE

The first step after data acquisition is as expected: **Preparation!** Before we can run the data threw a pipeline we have to convert the dicoms into niftis and execute the recon-all process for each subject. Otherwise we can't use the benefits of FreeSurfer. Because this can be done in rather little code it is a very good example to get started.

4.1 Convert Dicoms into Niftis

Let us start with the conversion of the 3D dicoms into 4D nifti files.

4.1.1 Import modules

The first step of every pipeline is always to import all the necessary modules. In this case we only need the basic modules `os`, `util`, `pipeline.engine` and the `freesurfer interface`.

```
import os                                    # system functions
import nipype.interfaces.freesurfer as fs    # freesurfer
import nipype.interfaces.utility as util     # utility
import nipype.pipeline.engine as pe          # pypeline engine
```

4.1.2 Define experiment specific parameters

After importing the modules, it is recommended to specify all the necessary variables like the path to the experiment folder, a list of the subject identifiers etc.

```
#Specification of the folder where the dicom-files are located at
experiment_dir = '~SOMEPATH/experiment'

#Specification of a list containing the identifier of each subject
subjects_list = ['subject1','subject2','subject3']

#Specification of the name of the dicom and output folder
dicom_dir_name = 'dicom' #if the path to the dicoms is: '~SOMEPATH/experiment/dicom'
data_dir_name = 'data'   #if the path to the data should be: '~SOMEPATH/experiment/data'
```

4.1.3 Define nodes to use

Let us now construct the three nodes we want to use to create our preparation pipeline. First we need to define the infosource node which specifies on which subjects the workflow is run on.

```
#Node: Infosource - we use IdentityInterface to create our own node, to specify
#                   the list of subjects the pipeline should be executed on
infosource = pe.Node(interface=util.IdentityInterface(fields=['subject_id']),
                                                      name="infosource")
infosource.iterables = ('subject_id', subjects_list)
```

Now we define the main node for the data conversion, DICOMConvert.

```
#Node: DICOMConvert - converts the .dcm files into .nii and moves them into
#                     the folder "data" with a subject specific subfolder
dicom2nifti = pe.Node(interface=fs.DICOMConvert(), name="dicom2nifti")
```

If we now look at the section nipype.interfaces.freesurfer.preprocess about the DICOMConvert node we see the following:

[Mandatory]

```
base_output_dir : (a directory name)
                  directory in which subject directories are created
dicom_dir : (an existing directory name)
            dicom directory from which to convert dicom files
```

Remember that the path to the dicoms changes from subject to subject. Therefore we can't declare the variable `dicom_dir` right away. But because we already have an `infosource` node, we will be able to overcome this problem when we establish the connections between the nodes. Let us first define the path to the output dir:

```
dicom2nifti.inputs.base_output_dir = experiment_dir + '/' + data_dir_name
#This will store the output to '~SOMEPATH/experiment/data'
```

Now we have to specify the **optional** inputs `file_mapping`, `out_type` and `subject_dir`.

```
dicom2nifti.inputs.file_mapping = [('nifti','*.nii'),('info','dicom.txt'),('dti','*dti.bv*')]
dicom2nifti.inputs.out_type = 'nii'
dicom2nifti.inputs.subject_dir_template = '%s'
```

This `dicom2nifti` node will convert the dicoms into niftis and create a summary text file calls **shortinfo.txt** which contains most of the important informations about the dicoms:

```
dcmdir ~SOMEPATH/experiment/dicom/subject1
PatientName subject1
StudyDate 20150403
StudyTime 083819.578000
1 fl2d1 localizer_BC 578000-1-1.dcm
2 fl2d1 localizer_32 578000-2-1.dcm
3 fl3d1_ns AAScout 578000-3-100.dcm
4 tfl3d1_ns T1_MPRAGE_1mm_iso 578000-4-100.dcm
5 epfid2d1_90 ge_functionals_2meas 578000-5-1.dcm
6 fm2d2r field_mapping 578000-6-10.dcm
7 fm2d2r field_mapping 578000-7-10.dcm
8 epir2d1_96 ep2d_t1w 578000-8-10.dcm
9 epfid2d1_96 ge_functionals_125 578000-9-10.dcm
10 epfid2d1_96 ge_functionals_125 578000-10-10.dcm
```

But because we also want to know how many time points were acquired and what the resolution of the dicoms are we'll have to run a node called `ParseDICOMDIR` which gives us an output text file with the following content:

```
  1 578000-1-2.dcm    1 0     2 0  512 512   3   1    0.0086   4.0000 localizer_BC
  4 578000-2-2.dcm    2 0     2 0  512 512   3   1    0.0086   4.0000 localizer_32
  7 578000-3-1.dcm    3 0     1 0  128 128 128   2    0.0024   1.1300 AAScout
263 578000-4-1.dcm    4 0     1 0  256 256 176   1    2.5300   3.4800 T1_MPRAGE_1mm_iso
439 578000-5-1.dcm    5 0     1 1   96  90  28   2    9.5000  30.0000 ge_functionals_2meas
441 578000-6-1.dcm    6 0     1 0   96  96  28   1    0.5000   2.8300 field_mapping
469 578000-7-1.dcm    7 0     1 0   96  96  28   1    0.5000   5.2900 field_mapping
497 578000-8-1.dcm    8 0     1 0   96  96  28   1   10.0000  56.0000 ep2d_t1w
```

```
525 578000-9-1.dcm    9 1     2 1   96  96  28  60    8.0000  30.0000 ge_functionals_125
585 578000-10-1.dcm  10 1     2 1   96  96  28  60    8.0000  30.0000 ge_functionals_125
```

This output shows us that the T1-file has resolution of 256x256x176 and that the two functional runs, file 9 and 10, have 60 time points. Now, let us implement this `ParseDICOMDir` node.

```
#Node ParseDICOMDIR - for creating a nicer nifti overview textfile
dcminfo = pe.Node(interface=fs.ParseDICOMDir(), name="dcminfo")
dcminfo.inputs.sortbyrun = True
dcminfo.inputs.summarize = True
dcminfo.inputs.dicom_info_file = 'nifti_overview.txt'
```

As before with the dicom2nifti node, we have the problem with the dcminfo node, that the input variable `dicom_dir` changes for each subject. And as before, we will see how to handle this issue during the connection of the nodes.

4.1.4 Define pipeline

After we've defined all the nodes we want to use, we are ready to implement the preparation pipeline:

```
#Initiation of the preparation pipeline
prepareflow = pe.Workflow(name="prepareflow")

#Define where the workingdir of the all_consuming_workflow should be stored at
prepareflow.base_dir = experiment_dir + '/workingdir_prepareflow'
```

4.1.5 Specify node connections

Now we've defined all the nodes and implemented the preparation pipeline. The only thing missing is the connection of the different nodes in the pipeline. But before we can do this, it is important to be aware about outputs, that get created and the inputs that have to be provided and more importantly how they look like.
infosource:
This node provides an output called `subject_id` that is either
`'subject1'`,
`'subject2'`,
`'subject3'`.

dicom2nifti:
This node needs `dicom_dir` as an input that is either:
`'~SOMEPATH/experiment/dicom/subject1/'`,
`'~SOMEPATH/experiment/dicom/subject2/'`,
`'~SOMEPATH/experiment/dicom/subject3/'`.

dcminfo:
This node needs `dicom_dir` as an input that is either:
`'~SOMEPATH/experiment/data/subject1/'`,
`'~SOMEPATH/experiment/data/subject2/'`,
`'~SOMEPATH/experiment/data/subject3/'`.

As you can see, for both nodes `dicom2nifti` and `dcminfo` the input from dicom_dir does change for each subject. But so also do the outputs of `infosource`. The only thing we have to do is to take the output `'subject1'` from `infosource` and change it into `'~SOMEPATH/experiment/dicom/subject1/'` for `dicom2nifti` and into `'~SOMEPATH/experiment/data/subject1/'` for `dcminfo`. This can be accomplished by inserting a function into the connection process.

Let's call this function `pathfinder`, which takes as arguments the subjectname and the foldername (either `dicom` or `data`) and returns `'~SOMEPATH/experiment/foldername/subjec_name/'`.

```
def pathfinder(subjectname, foldername):
    return os.path.join(experiment_dir, foldername, subjectname)
```

Now we can start with the connection of the nodes.

```
#Connect all components
prepareflow.connect([(infosource, dicom2nifti,[('subject_id', 'subject_id')]),
                     (infosource, dicom2nifti,[(('subject_id', pathfinder, dicom_dir_name),
                                                'dicom_dir')]),
                     (infosource, dcminfo,[(('subject_id', pathfinder, dicom_dir_name),
                                            'dicom_dir')]),
                     ])
```

Perhaps it's best if we take a closer look at the second connection. As it is written, the function `pathfinder` takes `'subject_id'` as its first argument which will represent `subjectname` in the function. `'dicom_dir_name'` will be given as the second argument `foldername` to the `pathfinder` function. The return value of the `pathfinder` function will than be sent as input `'dicom_dir'` to the `dicom2nifti` node.

4.1.6 Important for Nipype 0.4 users

There's an important issue about functions you have to consider in Nipype Version 0.4. If you would want to run the pathfinder function above you would encounter following error:

NameError: `("global name 'os' is not defined", 'Due to engine constraints all imports have to be done inside each function definition')`

This occurs because all values that you are using in a function have to be specified or imported within it's boundaries. Therefore we have to extend the function a bit by importing the `os` module and specifying the experiment_dir variable.

```
def pathfinder(subject, foldername):
    import os
    experiment_dir = '~SOMEPATH/experiment'
    return os.path.join(experiment_dir, foldername, subject)
```

4.1.7 Run pipeline

After the connection of the nodes there is only one last thing to do. To actually run the pipeline.

```
prepareflow.run(plugin='MultiProc', plugin_args={'n_procs' : 2})
```

Important: After running the prepareflow a folder called **data** gets created that contains all nifti files for each subject. For further work it is recommended to rename those files. An example would be to rename:
- `578000-4.nii` into `struct.nii` because it is the T1 file
- `578000-9.nii` into `func1.nii` because it is the first functional session file
- `578000-10.nii` into `func2.nii` because it is the second functional session file

4.1.8 Clean up (optional)

After running the prepareflow we now have a folder called **data** that contains the converted nifti files. Additionally the pipeline has created a folder **workingdir_prepareflow** which contains a lot of unnecessary outputs. But within this pile of data lie our `nifti_overview.txt` files for each subject, created by the `dcminfo node`. That's why I would recommend to add some additional python code after the run command to save those outputs in the corresponding subject subfolders in the data folder.

In our case the data we are interested in is in the folder `'~SOMEPATH/experiment/workingdir_prepareflow/prepareflow/_subject_id_subject1/dcminfo'` and we want to move it into

'/mindhive/gablab/u/mnotter/Desktop/TEST/data/subject1'. This can be accomplished with the following code:

```
#to run the loop for each subject
for subject in subjects_list:

    #specify where the nifti_overview.txt file is stored at
    from_path = os.path.join(prepareflow.base_dir,prepareflow.name,'_subject_id_%s'%subject,
                             dcminfo.name,dcminfo.inputs.dicom_info_file)

    #specify where to store the nifti_overview.txt file at
    to_path = os.path.join(dicom2nifti.inputs.base_output_dir,subject)

    #with os.system('text') you're able to state the command 'text' in your terminal
    #therefore we use mv to move the date
    os.system('mv %s %s'%(from_path, to_path))
```

Note: Following are the values of the variables we are using:

```
prepareflow.base_dir                = '~SOMEPATH/experiment/workingdir_prepareflow'
prepareflow.name                    = 'prepareflow'
'_subject_id_%s'%subject            = '_subject_id_subject1' for the first run
dcminfo.name                        = 'dcminfo'
dcminfo.inputs.dicom_info_file      = 'nifti_overview.txt'
dicom2nifti.inputs.base_output_dir  = '~SOMEPATH/experiment/data'
```

To finally delete the workingdirectory of the prepare pipeline with all it's content we can use the command `rm -rf ~SOMEPATH/experiment/workingdir_prepareflow` and we're done.

```
os.system('rm -rf %s'%prepareflow.base_dir)
```

Hint: The code to this section can be found here: dicom2nifti.py

4.2 Run Recon-All with Nipype

Now that we are able to build a preparation pipeline, it will be a piece of cake to create a recon-all pipeline. Only after executing the recon-all algorithm will we be able to use all the benefits of the FreeSurfer interface.

4.2.1 Prepare modules and important variables

First let us again import all necessary modules and specify the experiment specific parameters.

```
import os                                    # system functions
import nipype.interfaces.freesurfer as fs    # freesurfer
import nipype.interfaces.utility as util     # utility
import nipype.pipeline.engine as pe          # pypeline engine

#Specification of the folder where the dicom-files are located at
experiment_dir = '~SOMEPATH/experiment'

#Specification of a list containing the identifier of each subject
subjects_list = ['subject1','subject2','subject3']

#Specification of the output folder - where the T1 file can be found
data_dir_name = 'data'

#Node: SubjectData - we use IdentityInterface to create our own node, to specify
#      the list of subjects the pipeline should be executed on
infosource = pe.Node(interface=util.IdentityInterface(fields=['subject_id']),
                                                      name="infosource")
infosource.iterables = ('subject_id', subjects_list)
```

```
20
21 #Node: Recon-All - to generate surfaces and parcellations of structural
22 #                  data from anatomical images of a subject.
23 reconall = pe.Node(interface=fs.ReconAll(), name="reconall")
24 reconall.inputs.directive = 'all'
25
26 #Because the freesurfer_data folder doesn't exist yet
27 os.system('mkdir %s'%experiment_dir+'/freesurfer_data')
28
29 reconall.inputs.subjects_dir = experiment_dir + '/freesurfer_data'
30 T1_identifier = 'struct.nii' #This is the name we manually gave the T1-file
```

Important: If we don't create the `freesurfer_data` folder before we specify the `subjects_dir` variable we would get following error:
TraitError: `The 'subjects_dir' trait of a ReconAllInputSpec instance must be an existing directory name, but a value of '~SOMEPATH/experiment/freesurfer_data' <type 'str'> was specified.`

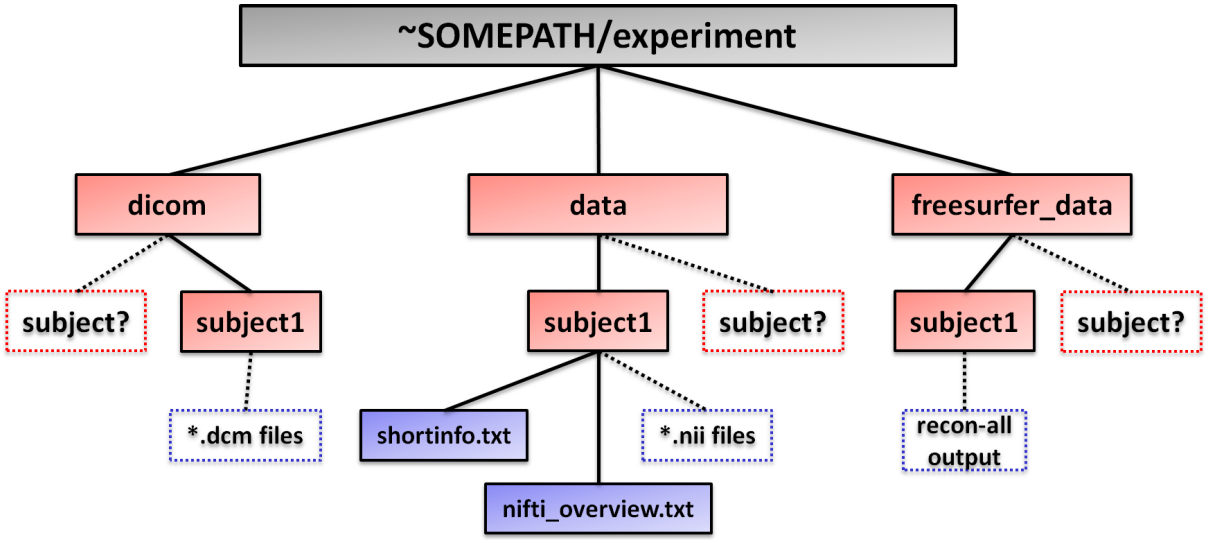
Now we are ready to implement the pipeline and connection the `infosource` with the `reconall` node. As in the example about the conversion of the dicoms into niftis, we will use again a function called `pathfinder` to specify the exact location of the T1-file of each subject. Note that the function takes three arguments in this case.

```
1  #implementation of the workflow
2  reconflow = pe.Workflow(name="reconflow")
3  reconflow.base_dir = experiment_dir + '/workingdir_reconflow'
4
5  #defenition of the pathfinder function
6  def pathfinder(subject, foldername, filename):
7      import os
8      experiment_dir = '~SOMEPATH/experiment/experiment'
9      return os.path.join(experiment_dir, foldername, subject, filename)
10
11 #connection of the nodes
12 reconflow.connect([(infosource, reconall,[('subject_id', 'subject_id')]),
13                    (infosource, reconall,[(('subject_id', pathfinder, data_dir_name,
14                                             T1_identifier),'T1_files')]),
15                    ])
16
17 #run the recon-all pipeline (as recommended in serial mode)
18 reconflow.run(plugin='Linear')
19
20 #to delete the workingdir of the reconflow we use again the shell-command "rm".
21 #The important recon-all files are already stored in the "freesurfer_data" folder
22 os.system('rm -rf %s'%reconflow.base_dir)
```

Hint: The code to this section can be found here: recon-all.py

4.3 Resulting Folder Structure

After we've prepared our data by converting the DICOM-files into NIFTI-files and running the recon-all process our folder structure should look like this:



In our parentfolder `~SOMEPATH/experiment` we have three main subfolders:

- **dicom** folder contains the raw DICOM-files of each subject in a subject named folder
- **data** folder is the result from the `prepareflow` and contains the NIFTI-files
- **freesurfer_data** folder is the result of the `reconflow` and contains the output from the recon-all process

CHAPTER

FIVE

HOW TO BUILD A FIRST LEVEL PIPELINE

In this part you will learn how to do create a simple **first level analysis pipeline**. This pipeline is just an example, it takes the raw nifti data, does some preprocessing (e.g. realignment, smoothing, etc.) and finally estimates the concrete model.

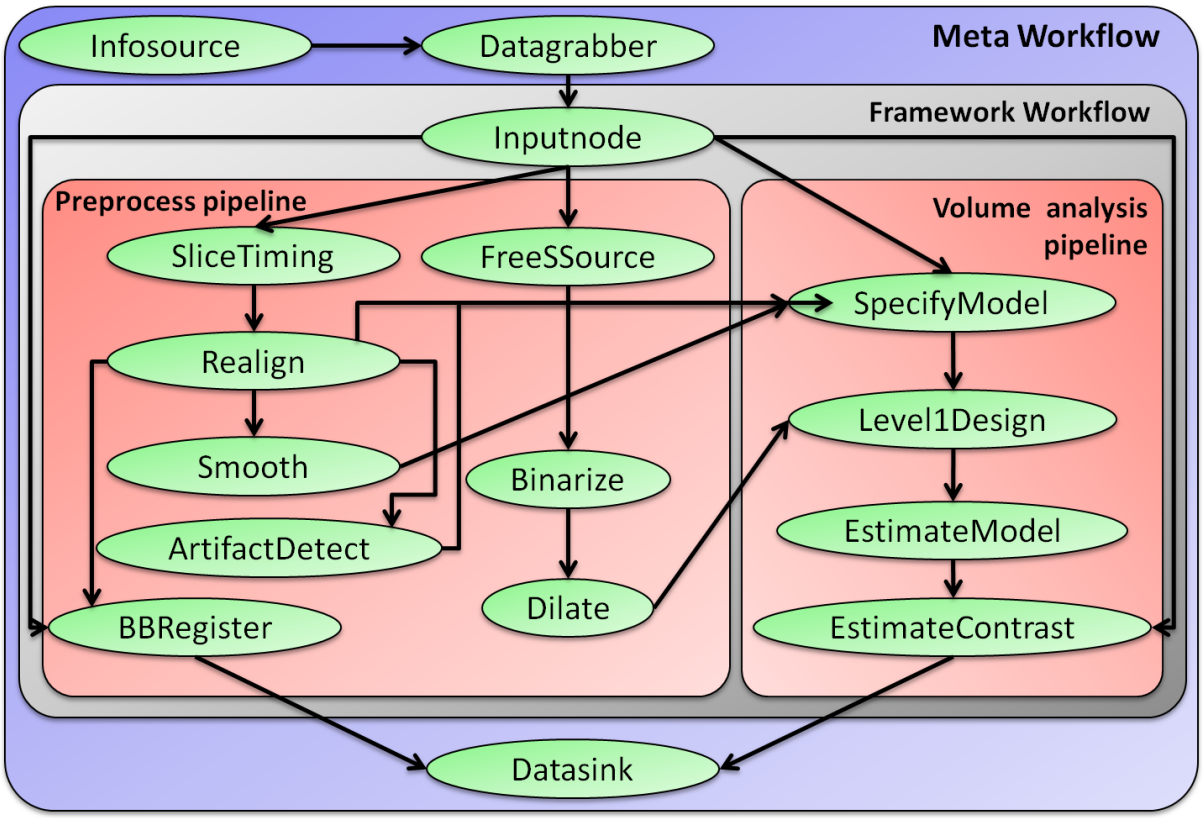
Note: The normalization of the results to a common template will not be done by this pipeline. A brief instruction on how to normalize your date with ANTS will be given in a latter chapter.

5.1 Define the structure of your pipeline

A typical fMRI workflow can be divided into two sections: **preprocessing** and **modeling**. The first one deals with cleaning data from confounds and noise and the second one fits a model to the cleaned data based on the experimental design. Now the best way to build your own pipeline is to think about the workflow the data should go through.

In this example we want to first run some preprocessing like `SliceTiming`, `Realignment`, `ArtifactDetection`, `BBRegister` and `Smoothing`. Additionally we also want to take the subject specific aseg file from the freesurfer folder, binarize it and dilate it to create a mask for the level1desing of the modeling part of the workflow. All this will be accommodate in a **preprocess pipeline**. After that we want to create a modeling part that estimates a concrete model by running first `SpecifyModel`, creating a `Level1Design`, `EstimateModel` and `EstimateContrast`. This will be done in a **volume analysis pipeline**.

The preprocess and the volume analysis pipeline together with the inputnode, will create the basic **framework workflow**. And this framework workflow will be connected to the infosource, the `DataGrabber` and the `DataSink` by a higher workflow, I'd like to call **meta workflow**. Visualized this pipeline would look like this:



The connections between the nodes will make more sense once we look at the inputs the given nodes need. Some may also ask why we haven't included the `Infosource`, `DataGrabber` and `DataSink` nodes into the `frameflow`? The reason is that those nodes highly dependent on the paradigm specific parameters and will change for every model/experiment. That's why we want to separate them from the framework workflow which will stay more or less the same for similar experiments. This does also give us the opportunity to just import this framework workflow into a new pipeline script.

5.2 Write your pipeline script

Now that we have defined how the structure of our pipeline and the connections between them should be we can start with writing the pipeline script.

5.2.1 Import modules

First we have to import all necessary modules.

```
import os                                    # system functions
import nipype.algorithms.modelgen as model   # model generation
import nipype.algorithms.rapidart as ra      # artifact detection
import nipype.interfaces.freesurfer as fs    # freesurfer
import nipype.interfaces.io as nio           # i/o routines
import nipype.interfaces.spm as spm          # spm
import nipype.interfaces.utility as util     # utility
import nipype.pipeline.engine as pe          # pypeline engine
import nipype.interfaces.base as base        # base routines
import nipype.interfaces.fsl.maths as math   #for dilating of the mask
```

5.2.2 Define experiment specific parameters

I suggest to keep things that change often between versions, models, experiments like subject names, output folders and name of functional runs at one place so that they can be accessed more easily in case you want to change them.

```
#To better access the parent folder of the experiment
experiment_dir = '~SOMEPATH/experiment'

#name of the subjects, functional files and output folders
subjects = ['subject1','subject2','subject3']
sessions = ['func1','func2']
nameOfLevel1Out = 'level1_output'

# Tell freesurfer what subjects directory to use
subjects_dir = experiment_dir + '/freesurfer_data'
fs.FSCommand.set_default_subjects_dir(subjects_dir)
```

Those specification mean that the name of the subjectfolders are "subject1", "subject2" and "subject3" and that each of those folders contain a "func1.nii" and "func2.nii" file which represents the nifti files for the first and the second functional run. But the exact structure of the folder will be later defined by the implementation of the `DataGrabber` node.

5.2.3 Define a pipeline for the preprocess

We first define the preprocess pipeline with a node for slicetiming, realignment, artifact detection, bbregister and smoothing.

```
#Initiation of the preprocess workflow
preproc = pe.Workflow(name='preproc')

#Node: Slicetiming
sliceTiming = pe.Node(interface=spm.SliceTiming(), name="sliceTiming")
sliceTiming.inputs.num_slices = 28
sliceTiming.inputs.time_repetition = 2.0
sliceTiming.inputs.time_acquisition = 2. - 2./28
sliceTiming.inputs.slice_order = range(1,28+1)      #for bottom up slicing
#sliceTiming.inputs.slice_order = range(28,0,-1)    #for top down slicing
sliceTiming.inputs.ref_slice = 1

#Node: Realign - for motion correction and to register all images to the mean image
realign = pe.Node(interface=spm.Realign(), name="realign")
realign.inputs.register_to_mean = True

#Node: Artifact Detection - to determine which of the images in the functional
#      series are outliers based on deviations in intensity or movement.
art = pe.Node(interface=ra.ArtifactDetect(), name="art")
art.inputs.norm_threshold       = 0.5
art.inputs.zintensity_threshold = 3
art.inputs.mask_type            = 'file'
art.inputs.parameter_source     = 'SPM'

#Node: BBRegister - to co-register the mean functional image generated by realign
#      to the subjects' surfaces.
bbregister = pe.Node(interface=fs.BBRegister(),name='bbregister')
bbregister.inputs.init = 'fsl'
bbregister.inputs.contrast_type = 't2'

#Node: Smooth - The volume smoothing option performs a standard SPM smoothing
volsmooth = pe.Node(interface=spm.Smooth(), name = "volsmooth")
volsmooth.inputs.fwhm = 6

```

```
#Node: FreeSurferSource - The get specific files from the freesurfer folder
fssource = pe.Node(interface=nio.FreeSurferSource(),name='fssource')
fssource.inputs.subjects_dir = subjects_dir

#Node: Binarize -  to binarize the aseg file for the dilation
binarize = pe.Node(interface=fs.Binarize(),name='binarize')
binarize.inputs.min = 0.5
binarize.inputs.out_type = 'nii'

#Node: DilateImage - to dilate the binarized aseg file and use it as a mask
dilate = pe.Node(interface=math.DilateImage(),name='dilate')
dilate.inputs.operation = 'max'
dilate.inputs.output_type = 'NIFTI'

#Connect up the preprocessing components
preproc.connect([(sliceTiming, realign,[('timecorrected_files', 'in_files')]),
                 (realign, bbregister,[('mean_image', 'source_file')]),
                 (realign, volsmooth, [('realigned_files', 'in_files')]),
                 (realign, art,[('realignment_parameters','realignment_parameters'),
                                ('mean_image', 'mask_file'),
                                ]),
                 (volsmooth,art,[('smoothed_files','realigned_files'),
                                 ]),
                 (fssource, binarize, [('aseg','in_file')]),
                 (binarize, dilate,[('binary_file','in_file')]),
                 (realign, art,[('realignment_parameters','realignment_parameters'),
                                ('mean_image','mask_file'),
                                ]),
                 ])
```

Note: If you are wondering how we knew which parameters to specify and which connections to establish. It is simple: Define or connect all mandatory inputs for each node. All the other optional inputs can be set as you please and more importantly as your model demands. For more informations about what is mandatory and what not, go to Interfaces and Algorithms.

5.2.4 Define a pipeline for the volume analysis

We than define the pipeline for the volume analysis with a node for model specification, first level design, parameter estimation and contrast estimation.

```
#Initiation of the volume analysis workflow
volanalysis = pe.Workflow(name='volanalysis')

#Node: SpecifyModel - Generate SPM-specific design information
modelspec = pe.Node(interface=model.SpecifySparseModel(), name= "modelspec")
modelspec.inputs.input_units = 'secs'
modelspec.inputs.time_repetition = 8.
modelspec.inputs.high_pass_filter_cutoff = 128
modelspec.inputs.model_hrf = True
modelspec.inputs.scale_regressors = True
modelspec.inputs.scan_onset = 4.
modelspec.inputs.stimuli_as_impulses = True
modelspec.inputs.time_acquisition = 2.
modelspec.inputs.use_temporal_deriv = False
modelspec.inputs.volumes_in_cluster = 1

#Node: Level1Design - Generate a first level SPM.mat file for analysis
level1design = pe.Node(interface=spm.Level1Design(), name= "level1design")
level1design.inputs.bases              = {'hrf':{'derivs': [0,0]}}
#level1design.inputs.bases              = {'fir':{'length':3, 'order' : 1}}
level1design.inputs.timing_units       = 'secs'
```

```
level1design.inputs.interscan_interval = modelspec.inputs.time_repetition

#Node: EstimateModel - to determine the parameters of the model
level1estimate = pe.Node(interface=spm.EstimateModel(), name="level1estimate")
level1estimate.inputs.estimation_method = {'Classical' : 1}

#Node: EstimateContrast - to estimate the first level contrasts we define later
contrastestimate = pe.Node(interface = spm.EstimateContrast(), name="contrastestimate")

#Connect up the volume analysis components
volanalysis.connect([(modelspec,level1design,[('session_info','session_info')]),
                     (level1design,level1estimate,[('spm_mat_file','spm_mat_file')]),
                     (level1estimate,contrastestimate,[('spm_mat_file','spm_mat_file'),
                                                       ('beta_images','beta_images'),
                                                       ('residual_image',
                                                        'residual_image')]),
                     ])
```

5.2.5 Define a framework workflow that contains the preprocess and the volume analysis

As we planed at the beginning we now want to integrate those two pipelines into a bigger framework workflow and add an inputnode that feeds these pipelines with parameters.

```
#Initiation of the framework workflow
frameflow = pe.Workflow(name='frameflow')

#Node: Inputnode - For this workflow the only necessary inputs are the functional
#      images, a freesurfer subject id corresponding to recon-all processed data,
#      the session information for the functional runs and the contrasts to be evaluated.
inputnode = pe.Node(interface=util.IdentityInterface(fields=['func','subject_id',
                                                             'session_info','contrasts']),
                    name='inputnode')

#Connect up the components into an integrated workflow.
frameflow.connect([(inputnode,preproc,[('func','sliceTiming.in_files'),
                                       ('subject_id','bbregister.subject_id'),
                                       ('subject_id','fssource.subject_id'),
                                       ]),
                   (inputnode, volanalysis,[('session_info','modelspec.subject_info'),
                                            ('contrasts','contrastestimate.contrasts'),
                                            ]),
                   (preproc, volanalysis, [('realign.realignment_parameters',
                                            'modelspec.realignment_parameters'),
                                           ('volsmooth.smoothed_files',
                                            'modelspec.functional_runs'),
                                           ('art.outlier_files',
                                            'modelspec.outlier_files'),
                                           ('dilate.out_file','level1design.mask_image'),
                                           ])
                   ])
```

5.2.6 Define infosource, datagrabber and datasink

We now have to create the `Infosource` that defines the subjectlist, the `DataGrabber` that grabs the inputs and the `DataSink` which defines where we want to store the important outputs at.

```
#Node: Infosource - we use IdentityInterface to create our own node, to specify
#      the list of subjects the pipeline should be executed on
infosource = pe.Node(interface=util.IdentityInterface(fields=['subject_id']),
```

```
                      name="infosource")
infosource.iterables = ('subject_id', subjects)

#Node: DataGrabber - To grab the input data
datasource = pe.Node(interface=nio.DataGrabber(infields=['subject_id'],
                                               outfields=['func', 'struct']),
                      name = 'datasource')

#Define the main folder where the data is stored at and define the structure of it
datasource.inputs.base_directory = experiment_dir
datasource.inputs.template = 'data/%s/%s.nii'

info = dict(func=[['subject_id', sessions]],
            struct=[['subject_id','struct']])

datasource.inputs.template_args = info

#Node: Datasink - Create a datasink node to store important outputs
datasink = pe.Node(interface=nio.DataSink(), name="datasink")
datasink.inputs.base_directory = experiment_dir

#Define where the datasink input should be stored at
datasink.inputs.container = 'results/' + nameOfLevel1Out
```

5.2.7 Define contrasts and model specification

We now set up the model specific components like contrasts, names of conditions, parametric modulators onset, duration of a trial etc.

```
#Names of the conditions
namesOfConditions = ['basic','condition1','condition2','condition3']

#Define different contrasts
cont1 = ('basic vs. conditions','T', namesOfConditions, [3,-1,-1,-1])
cont2 = ('all vs. condition1',  'T', namesOfConditions, [0,1,0,0])
cont3 = ('all vs. condition2',  'T', namesOfConditions, [0,0,1,0])
cont4 = ('all vs. condition3',  'T', namesOfConditions, [0,0,0,1])
cont5 = ('session1 vs session2','T', namesOfConditions, [1,1,1,1],[1,-1])

#store all contrasts into a list...
contrasts = [cont1, cont2, cont3, cont4, cont5]

#...and feed those contrasts to the inputnode filed 'contrasts'
frameflow.inputs.inputnode.contrasts = contrasts

#Function: Subjectinfo - This function returns subject-specific information about
#          the experimental paradigm. This is used by the SpecifyModel function
#          to create the information necessary to generate an SPM design matrix.
def subjectinfo(subject_id):

    from nipype.interfaces.base import Bunch

    namesOfConditions = ['basic','condition1','condition2','condition3']

    #Onset Times in seconds
    onsetTimes = [[1,10,42,49.6,66.1,74.1,97.6,113.6,122.2,130.2,137.2,153.7,169.2,
                   185.7,201.8,290.4,313.4,321.4,377.5,401.5,410,418.6,442.1,473.6],
                  [17.5,82.1,89.6,145.2,225.3,242.3,281.4,426.6],
                  [26,162.2,209.3,249.3,265.9,205.4,450.1,386],
                  [34,273.4,329.5,338.5,354,362,370,466.4]
                  ]
```

```
33
34      #Define the parametric modulators
35      para_modu = [None,
36                   Bunch(name=['target2','target3'], poly=[[1],[1]],
37
38                         param = [[0,0,1,0,0,0,0,0],[0,0,0,0,1,0,0,1]]),
39                   Bunch(name=['target2','target3'], poly=[[1],[1]],
40                         param = [[0,0,0,1,1,1,0,0],[1,0,0,0,0,0,1,1]]),
41                   Bunch(name=['target2','target3'], poly=[[1],[1]],
42                         param = [[0,1,0,0,0,1,0,1],[0,0,0,0,1,0,1,0]]),
43                   ]
44
45      output = []
46
47      #We add the model specific parameters twice to the output list because we
48      #have 2 functional runs which were performed identical.
49      for r in range(2):
50          output.append(Bunch(conditions=namesOfConditions,
51                              onsets=onsetTimes,
52                              durations=[[2] for s in namesOfConditions],
53                              amplitudes=None,
54                              tmod=None,
55                              pmod=para_modu,
56                              regressor_names=None,
57                              regressors=None))
58      return output  #this output will later be returned to inputnode.session_info
```

5.2.8 Define the meta pipeline

After setting up all nodes, parameters and subpipelines, we now want to create the pipeline that contains everything. The one that gets executed at the end.

```
1   #Initiation of the metaflow
2   metaflow = pe.Workflow(name="metaflow")
3
4   #Define where the workingdir of the metaflow should be stored at
5   metaflow.base_dir = experiment_dir + '/results/workingdir'
6
7   #Connect up all components
8   metaflow.connect([(infosource, datasource,[('subject_id', 'subject_id')]),
9                     (datasource,frameflow,[('func','inputnode.func')]),
10                    (infosource,frameflow,[('subject_id','inputnode.subject_id'),
11                                           (('subject_id', subjectinfo),
12                                             'inputnode.session_info'),
13                                           ]),
14                    (frameflow,datasink,[('preproc.bbregister.out_reg_file',
15                                          'bbregister'),
16                                         ('volanalysis.contrastestimate.spm_mat_file',
17                                          'vol_contrasts.@spm_mat'),
18                                         ('volanalysis.contrastestimate.spmT_images',
19                                          'vol_contrasts.@T'),
20                                         ('volanalysis.contrastestimate.con_images',
21                                          'vol_contrasts.@con'),
22                                         ])
23                    ])
```

Note: Some may wonder what the `@spm_mat`, `@T` and `@con` cause in the connection between the **frameflow** and the **datasink**. This specification means that all the `spm_mat_file`, `spmT_images` and `con_images` files all get saved into the same **datasink** folder with the name `vol_contrasts`. The `.@id` just specifies an identifier for the saving process.

5.3 Adding a surface analysis to your pipeline

Running the first level analysis also on the surface, and not just on the volume, can be achieved with just little more code. Because the **surface analysis pipeline** is similar to the one on the volume we can just clone the **volume analysis pipeline**:

```
#creates a clone of volanaylsis called surfanalysis
surfanalysis = volanalysis.clone(name='surfanalysis')
```

Now we have to integrate this surfanalysis into the `frameflow` and into the `metaflow`. The connections are almost the same as they are for the **volume analysis pipeline**. In this example the only difference is that we don't want to use smoothed data for the analysis on the surface because we will be smoothing the data latter on the second level surface analysis.

This means instead of taking the smoothed files from the preprocess pipeline and feeding them to the `modelspec` node we will take the files directly from the `realign` node.

```
#integration  of the surfanalysis into the frameflow
frameflow.connect([(inputnode, surfanalysis,[('session_info','modelspec.subject_info'),
                                             ('contrasts','contrastestimate.contrasts'),
                                             ]),
                   (preproc, surfanalysis, [('realign.realignment_parameters',
                                             'modelspec.realignment_parameters'),
                                            ('realign.realigned_files',
                                             'modelspec.functional_runs'),
                                            ('art.outlier_files',
                                             'modelspec.outlier_files'),
                                            ('dilate.out_file','level1design.mask_image'),
                                            ]),
                   ])

#integration  of the surfanalysis into the metaflow
metaflow.connect([(frameflow,datasink,[('surfanalysis.contrastestimate.spm_mat_file',
                                        'surf_contrasts.@spm_mat'),
                                       ('surfanalysis.contrastestimate.spmT_images',
                                        'surf_contrasts.@T'),
                                       ('surfanalysis.contrastestimate.con_images',
                                        'surf_contrasts.@con'),
                                       ])
                   ])
```

5.3.1 Run the pipeline and generate the graph

Finally, after everything is set up correctly we can run the pipeline and additionally let it draw the two graphs.

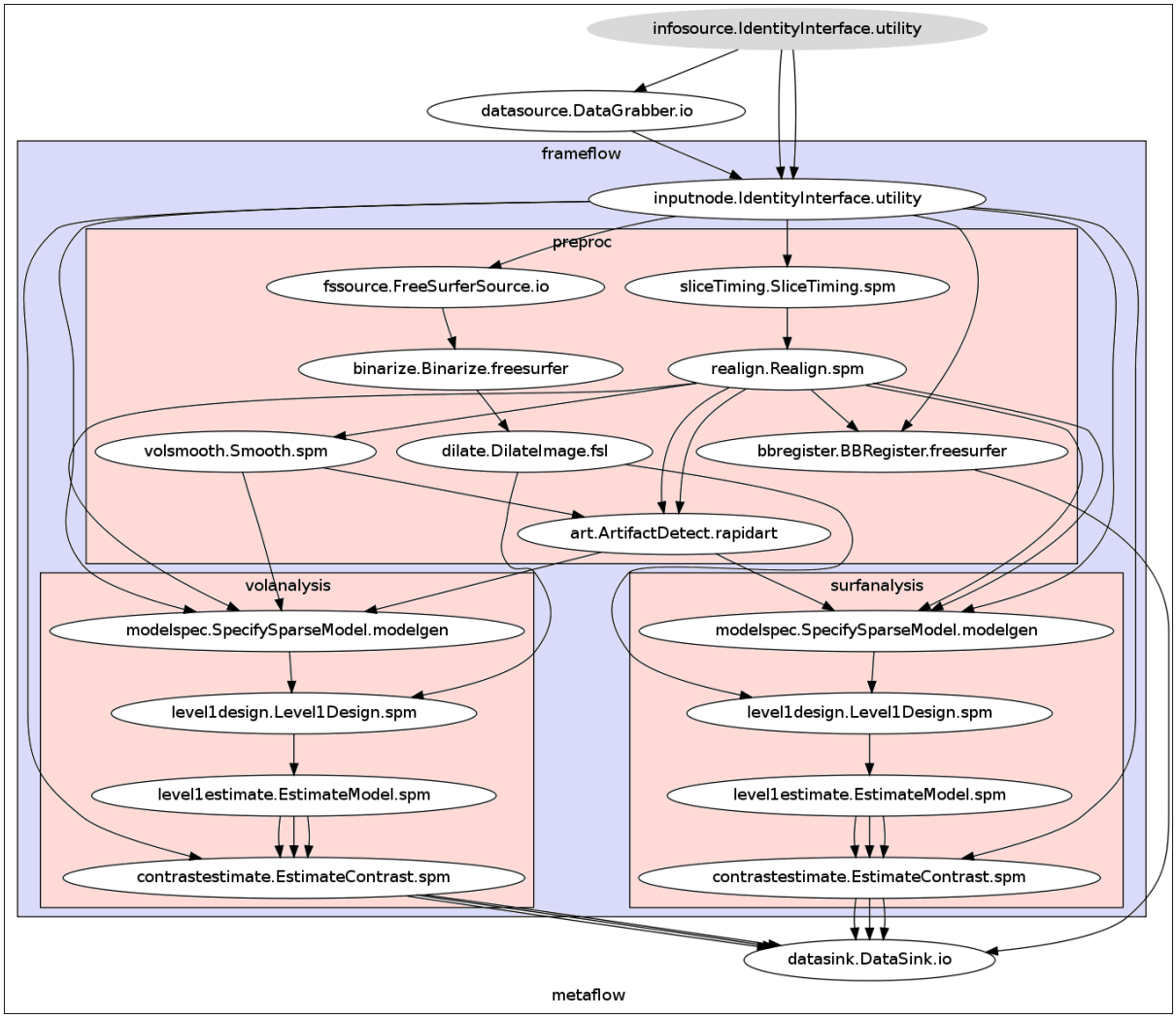
```
#Run the analysis pipeline and create the two graphs that visually represent the workflow.
metaflow.write_graph(graph2use='flat')
metaflow.run(plugin='MultiProc', plugin_args={'n_procs' : 2})
```

Hint: The code for a first level analysis on the volume and the surface can be found here: firstlevelpipeline.py

5.4 Visualize your pipeline

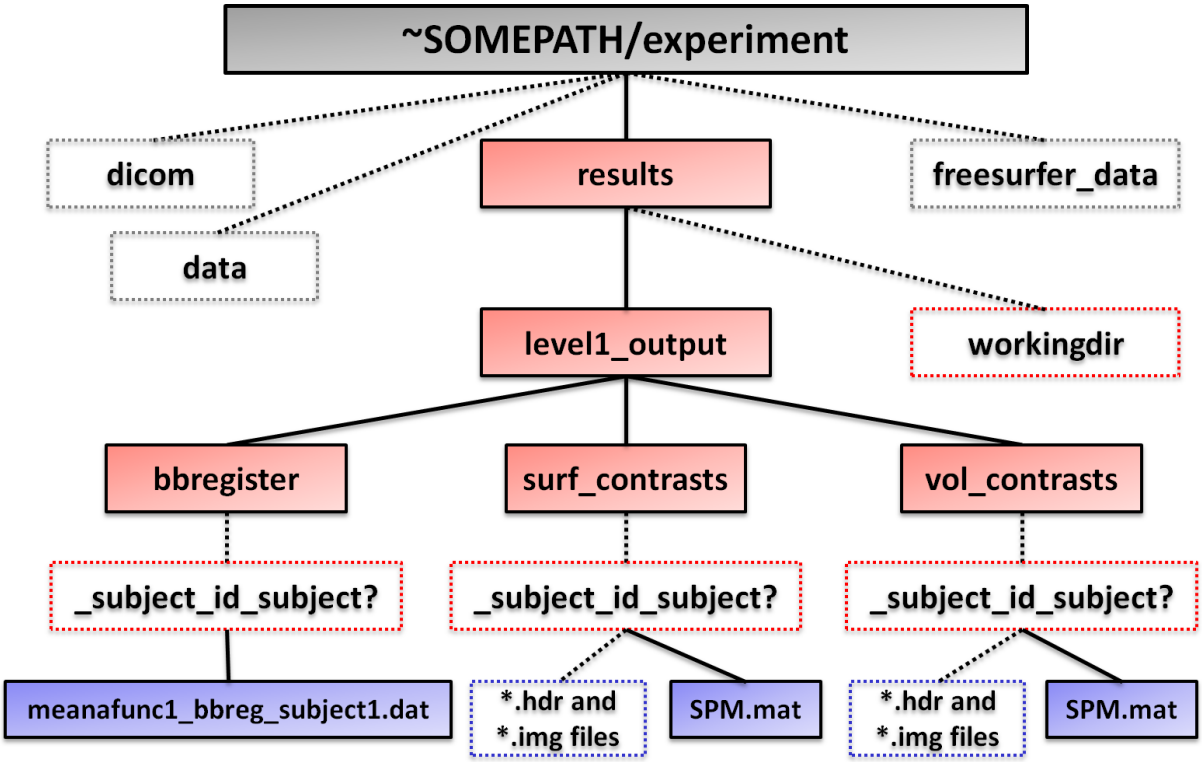
The visualization of this graph can be seen in all versions in chapter **How To Visualize A Pipeline** under example graphs.

But still, here's the **hierarchical graph** for the `metaflow`:



5.5 Resulting Folder Structure

After we've run our **first level analysis pipeline** on the volume and the surface our folder structure should look like this:



Additionally to the original `dicom`, `data` and `freesurfer_data` folder we now have a new folder called `results` that contains:

- **workingdir** folder that contains all the data that gets created from the `metaflow` pipeline. Because this is the highest storage consumer and to save space it is **highly recommended** to delete this folder as soon as possible.
- **level1_output** folder which is the **datasink** of the `metaflow`. It contains:
  - **bbregister** folder with the bbregister file for each subject
  - **surf_contrasts** folder with the estimated surface contrasts and the SPM.mat file for each subject
  - **vol_contrasts** folder with the estimated volume contrasts and the SPM.mat file for each subject

CHAPTER

SIX

HOW TO BUILD A SECOND LEVEL PIPELINE

In this part you will learn how to do a **second level analysis** on the volume and the surface. We want combine both pipelines in a common parent workflow. Of course this would be possible, but it is also a advantage to be able to run the pipelines separately.

Note: Before we can run the **second level analysis** on the volume we have to normalize the estimated volume contrasts from the first level pipeline into a common subject space. One method how you can achieve this will be covered in the chapter about ANTS. This normalization isn't required for a **second level analysis** on the surface. Because the second level surface analysis will be done with FreeSurfer, we can use the subject specific informations won from the recon-all process.

6.1 Specify condition independent parameters

It doesn't matter if you are running you're analysis only on the volume, on the surface or both. As always we're beginning by importing the necessary modules and implementation of the experiment specific parameters.

6.1.1 Import modules

```
import nipype.interfaces.freesurfer as fs # freesurfer
import nipype.interfaces.io as nio        # i/o routines
import nipype.interfaces.spm as spm       # spm
import nipype.interfaces.utility as util  # utility
import nipype.pipeline.engine as pe       # pypeline engine
```

6.1.2 Define experiment specific parameters

```
#to better access the parent folder of the experiment
experiment_dir = '~SOMEPATH/experiment'

#tell FreeSurfer where the recon-all output is at
freesurfer_dir = experiment_dir + '/freesurfer_data'
fs.FSCommand.set_default_subjects_dir(freesurfer_dir)

#list of subjectnames
subjects = ['subject1', 'subject2', 'subject3']

#second level analysis pipeline specific components
nameOfLevel2Out = 'level2_output'
numberOfContrasts = 5 #number of contrasts you specified in the first level analysis
contrast_ids = range(1,numberOfContrasts+1) #to create a list with value [1,2,3,4,5]
```

Note: If you want to only analyze the 3rd and 4th contrasts, you could specify `contrast_ids` as `range(3,5)` which would create the list `[3,4]`

6.2 Second level analysis on the volume

6.2.1 Grab the data

As always we first have to specify where the `DataGrabber` node can find the data. Le's assume that we have stored our normalized estimated volume contrasts at: `'~SOMEPATH/experiment/results/level1_output/normcons/subject_name/'`.
This means the third contrast of the second subject would be at: `'~SOMEPATH/experiment/results/level1_output/normcons/subject2/con_0001.nii'`

```
#Node: DataGrabber - to collect all the con images for each contrast
l2volSource = pe.Node(nio.DataGrabber(infields=['con']), name="l2volSource")
path2normcons = experiment_dir + '/results/level1_output/normcons/subject*/con_%04d.nii'
l2volSource.inputs.template = path2normcons
l2volSource.iterables = [('con',contrast_ids)] # iterate over all contrast images
```

You might be confused with the asterisk in `l2volSource.inputs.template` and that we didn't iterated over the subjects. This is because of the nature of a second level analysis. We take all estimated contrasts of each subject at once into the analysis, therefore the asterisk. We only have to iterate the conditions e.g. the number of the contrasts.

6.2.2 Define nodes

```
#Node: OneSampleTTest - to perform an one sample t-test analysis
oneSampleTTestVolDes = pe.Node(interface=spm.OneSampleTTestDesign(),
                               name="oneSampleTTestVolDes")

#Node: EstimateModel - to estimate the model
l2estimate = pe.Node(interface=spm.EstimateModel(), name="l2estimate")
l2estimate.inputs.estimation_method = {'Classical' : 1}

#Node: EstimateContrast - to estimate the contrast (in this example just one)
l2conestimate = pe.Node(interface = spm.EstimateContrast(), name="l2conestimate")
cont1 = ('Group','T', ['mean'],[1])
l2conestimate.inputs.contrasts = [cont1]
l2conestimate.inputs.group_contrast = True

#Node: Threshold - to threshold the estimated contrast
l2threshold = pe.Node(interface = spm.Threshold(), name="l2threshold")
l2threshold.inputs.contrast_index = 1
l2threshold.inputs.use_fwe_correction = False
l2threshold.inputs.use_topo_fdr = True
l2threshold.inputs.extent_threshold = 1
#voxel threshold
l2threshold.inputs.extent_fdr_p_threshold = 0.05
#cluster threshold (value is in -ln()): 1.301 = 0.05; 2 = 0.01; 3 = 0.001,
l2threshold.inputs.height_threshold = 3
```

Note: Of course you can all different kinds of statistical analysis. Another example beside a one sample t-test would be multiple regression analysis. Such a node would look this:

```
#Node: MultipleRegressionDesign - to perform a multiple regression analysis
multipleRegDes = pe.Node(interface=spm.MultipleRegressionDesign(),
                         name="multipleRegDes")
#regressor1 and regressor2 for 3 subjects
multipleRegDes.inputs.covariates = [dict(vector=[-0.30,0.52,1.75],
```

```
                                        name='nameOfRegressor1'),
                                   dict(vector=[1.55,-1.80,0.77],
                                        name='nameOfRegressor2')]
```

6.2.3 Establish a second level volume pipeline

```
#Create 2-level vol pipeline and connect up all components
l2volflow = pe.Workflow(name="l2volflow")
l2volflow.base_dir = experiment_dir + '/results/workingdir_l2vol'
l2volflow.connect([(l2volSource,oneSampleTTestVolDes,[('outfiles','in_files')]),
                   (oneSampleTTestVolDes,l2estimate,[('spm_mat_file','spm_mat_file')]),
                   (l2estimate,l2conestimate,[('spm_mat_file','spm_mat_file'),
                                              ('beta_images','beta_images'),
                                              ('residual_image','residual_image')
                                              ]),
                   (l2conestimate,l2threshold,[('spm_mat_file','spm_mat_file'),
                                               ('spmT_images','stat_image'),
                                               ]),
                   ])
```

6.3 Second level analysis on the surface

An important difference between the format of the volume and the surface data is, that the surface data are img-files and are separated for both hemispheres. In contrast the volume data is in nifti-files which contain both hemispheres. This separation means that we have to iterate over the left ('lh') and the right ('rh') hemisphere.

6.3.1 Grab the data

As mentioned above, our **second level surface pipeline** does have to iterate over the different contrasts **and** the left and right hemisphere. This can be done with the usual individually defined `IdentityInterface` node.

```
#Node: IdentityInterface - to iterate over contrasts and hemispheres
l2surfinputnode = pe.Node(interface=util.IdentityInterface(fields=['contrasts','hemi']),
                          name='l2surfinputnode')
l2surfinputnode.iterables = [('contrasts', contrast_ids),
                             ('hemi', ['lh','rh'])]
```

Again we have to be aware about the structure of our data folder. We know from the **first level analysis pipeline** that our estimated surface contrasts are stored at: `'~SOMEPATH/experiment/result/level1_output/'`. This will be defined as `base_directory` of the datagrabber node.

```
#Node: DataGrabber - to collect contrast images and registration files
l2surfSource = pe.Node(interface=nio.DataGrabber(infields=['con_id'],
                                                 outfields=['con','reg']),
                       name='l2surfSource')
l2surfSource.inputs.base_directory = experiment_dir + '/level1_output/'
l2surfSource.inputs.template = '*'
l2surfSource.inputs.field_template = dict(con='surf_contrasts/_subject_id_*/con_%04d.img',
                                          reg='bbregister/_subject_id_*/*.dat')
l2surfSource.inputs.template_args = dict(con=[['con_id']],reg=[[]])
```

6.3.2 Define nodes

Now that we have defined where our data comes from, we can start with implementing the nodes.

```
#Node: Merge - to merge contrast images and registration files
merge = pe.Node(interface=util.Merge(2, axis='hstack'),name='merge')

#function to create a list of all subjects and the location of their specific files
def ordersubjects(files, subj_list):
    outlist = []
    for subject in subj_list:
        for subj_file in files:
            if '/_subject_id_%s/'%subject in subj_file:
                outlist.append(subj_file)
                continue
return outlist

#Node: MRISPreproc - to concatenate contrast images projected to fsaverage
concat = pe.Node(interface=fs.MRISPreproc(), name='concat')
concat.inputs.target = 'fsaverage'
concat.inputs.fwhm = 5  #the smoothing of the surface data happens here

#function that transforms a given list into tuples
def list2tuple(listoflist):
    return [tuple(x) for x in listoflist]

#Node: OneSampleTTest - to perform a one sample t-test on the surface
oneSampleTTestSurfDes = pe.Node(interface=fs.OneSampleTTest(),
                                name='oneSampleTTestSurfDes')
```

6.3.3 Establish a second level surface pipeline

```
#Create 2-level surf pipeline and connect up all components
l2surfflow = pe.Workflow(name='l2surfflow')
l2surfflow.base_dir = experiment_dir + '/results/workingdir_l2surf'
l2surfflow.connect([(l2surfinputnode,l2surfSource,[('contrasts','con_id')]),
                    (l2surfinputnode,concat,[('hemi','hemi')]),
                    (l2surfSource,merge,[(('con', ordersubjects, subjects),'in1'),
                                         (('reg', ordersubjects, subjects),'in2')]),
                    (merge,concat,[(('out', list2tuple),'vol_measure_file')]),
                    (concat,oneSampleTTestSurfDes,[('out_file','in_file')]),
                    ])
```

6.4 Store results in a common Datasink

If you want to store the data from the second level volume and surface pipeline at a common location you should use a `datasink` node. You can use the same `datasink` node for both pipelines because you aren't running both pipelines in the same run.

```
#Node: Datasink - Create a datasink node to store important outputs
l2datasink = pe.Node(interface=nio.DataSink(), name="l2datasink")
l2datasink.inputs.base_directory = experiment_dir + '/results'
l2datasink.inputs.container = nameOfLevel2Out

#integration of the datasink into the volume analysis pipeline
l2volflow.connect([(l2conestimate,l2datasink,[('spm_mat_file','l2vol_contrasts.@spm_mat'),
                                              ('spmT_images','l2vol_contrasts.@T'),
                                              ('con_images','l2vol_contrasts.@con'),
                                              ]),
                   (l2threshold,l2datasink,[('thresholded_map',
                                             'vol_contrasts_thresh.@threshold')]),
                   ])

```

```
#integration of the datasink into the surface analysis pipeline
l2surfflow.connect([(oneSampleTTestSurfDes,l2datasink,[('sig_file',
                                                        'l2surf_contrasts.@sig_file')])])
```

6.5 Run pipelines

Now that we have set up both pipelines we are able to run them. Note that those lines of codes mean that the **second level surface pipeline** won't be started before the **second level volume pipeline** has terminated.

```
l2volflow.write_graph(graph2use='flat')
l2volflow.run(plugin='MultiProc', plugin_args={'n_procs' : 2})

l2surfflow.write_graph(graph2use='flat')
l2surfflow.run(plugin='MultiProc', plugin_args={'n_procs' : 2})
```

Now that everything has been executed you are able to take a look at your results. I recommend to use the following tools to use:

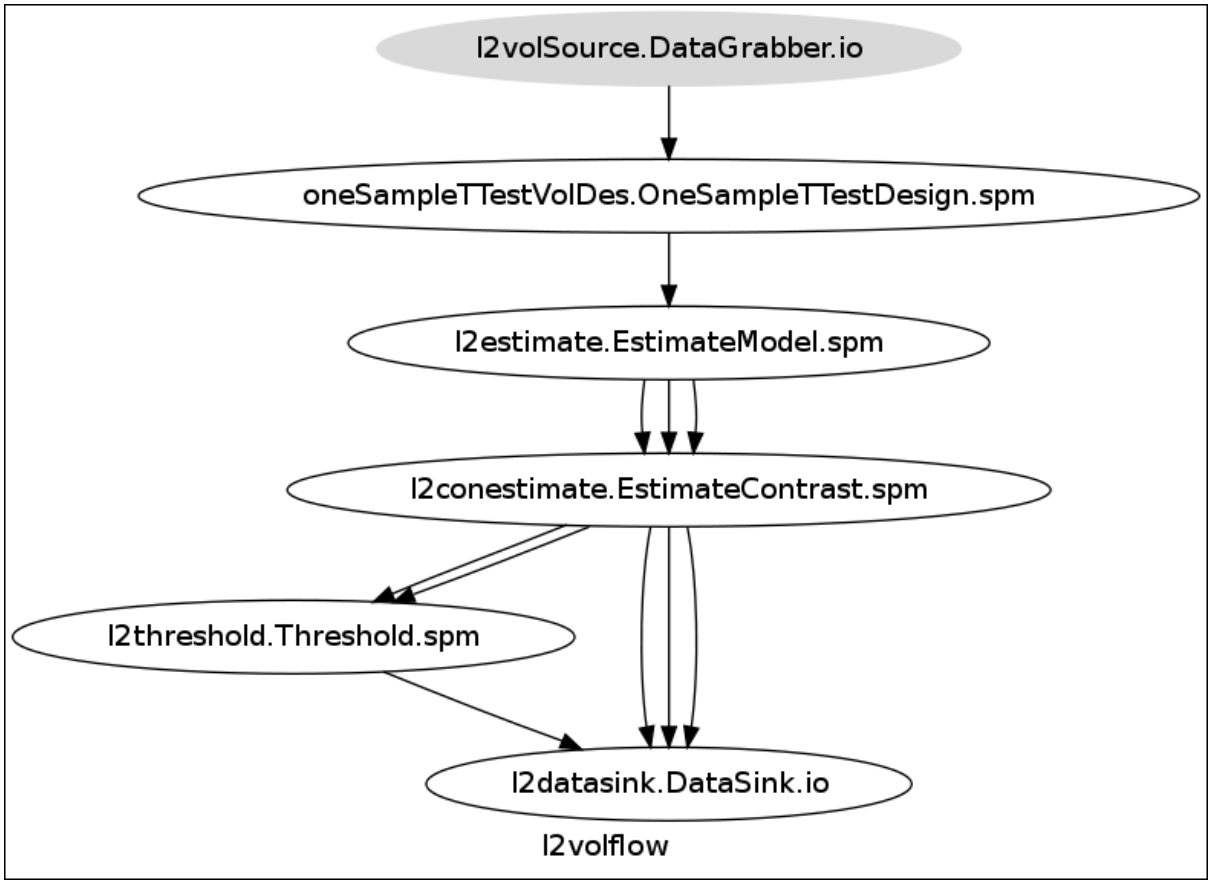
- **Matlab's SPM fmri** for the results from the contrast estimation on the volume stored in datasinkfolder `l2vol_contrasts`
- **FreeSurfer's Freeview** for the results from the threshold node stored in datasinkfolder `l2vol_contrasts_thresh`. Overlay the results on your normalization brain template
- **FreeSurfer's Freeview** for the results from the contrast estimation on the surface stored in datasinkfolder `l2surf_contrasts`. Overlay the results on FreeSurfer's normalization brain template (e.g. fsaverage)

Hint: The code for the second level pipeline on the volume and on the surface can be found here: secondlevelpipeline.py

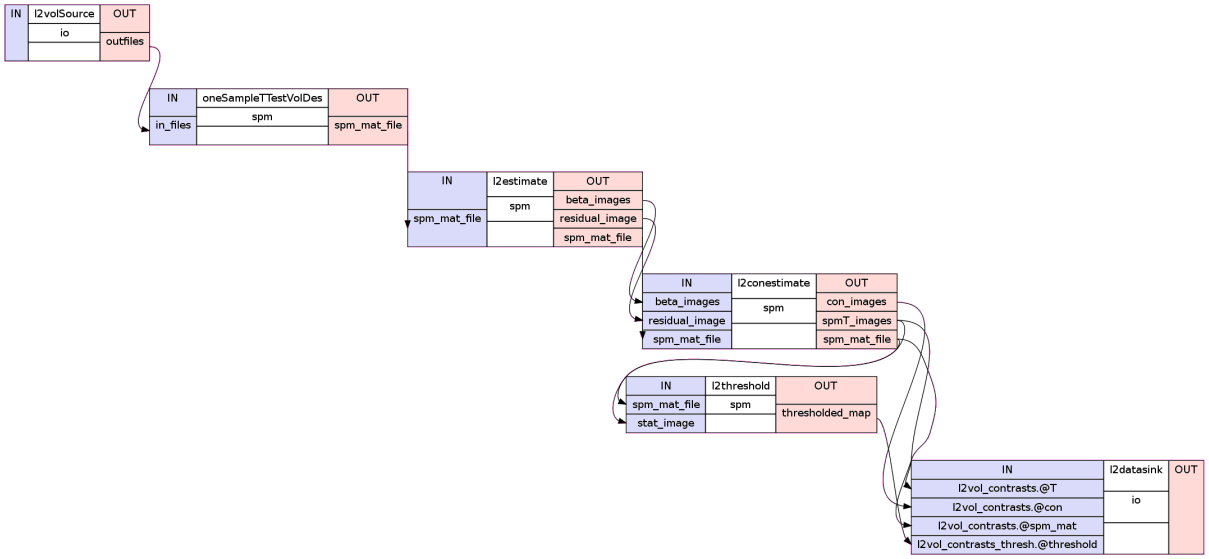
6.6 Visualize pipelines

6.6.1 Second Level Volume Pipeline

This graph of the `hierarchical` version shows the `l2volflow`:

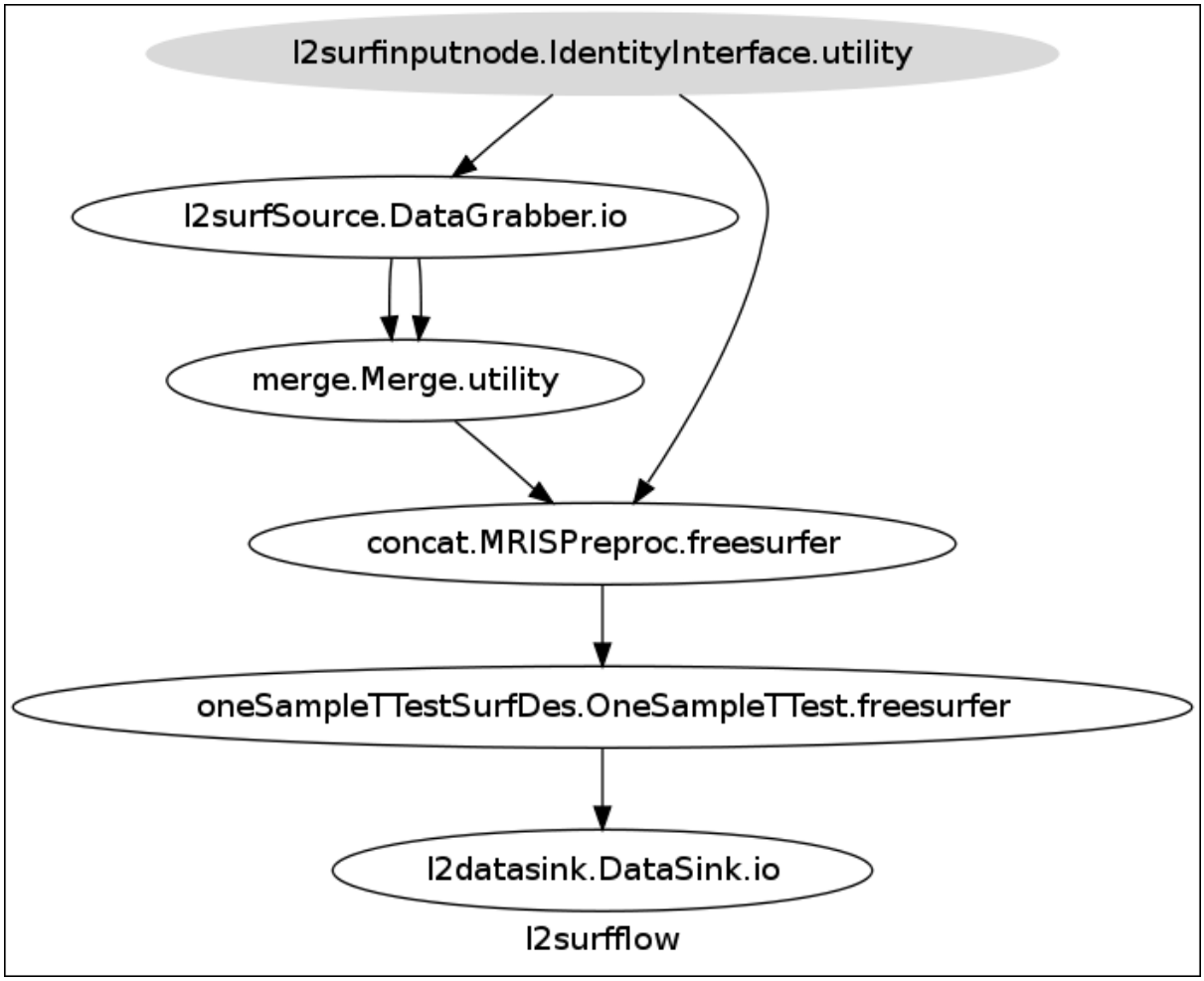


This detailed graph of the `flat` version shows the `l2volflow`:

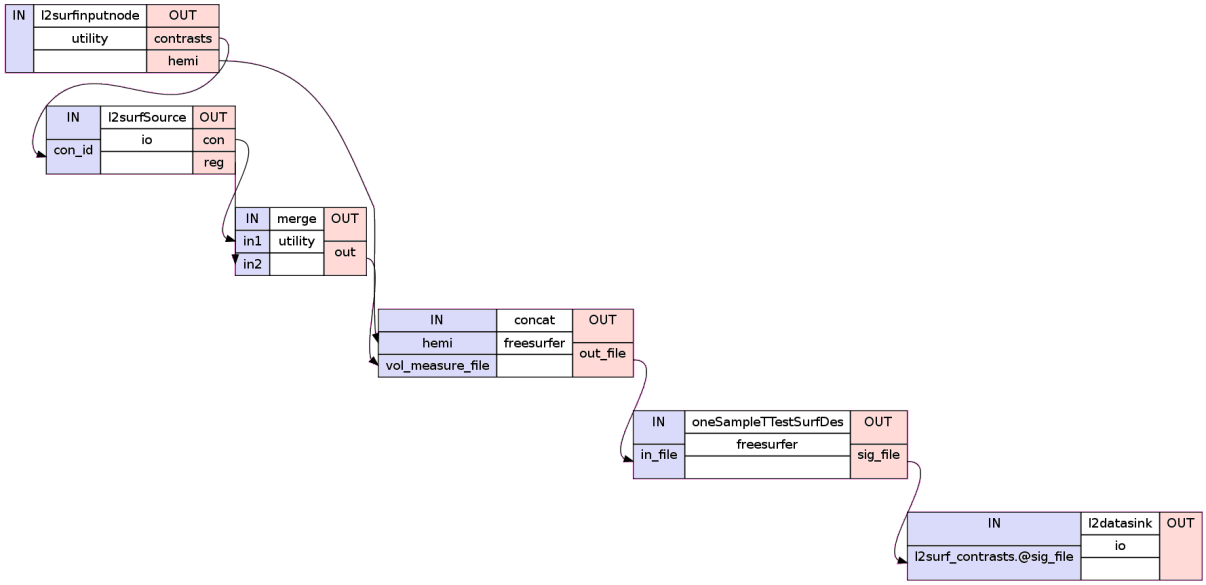


6.6.2 Second Level Surface Pipeline

This graph of the `hierarchical` version shows the `l2surfflow`:

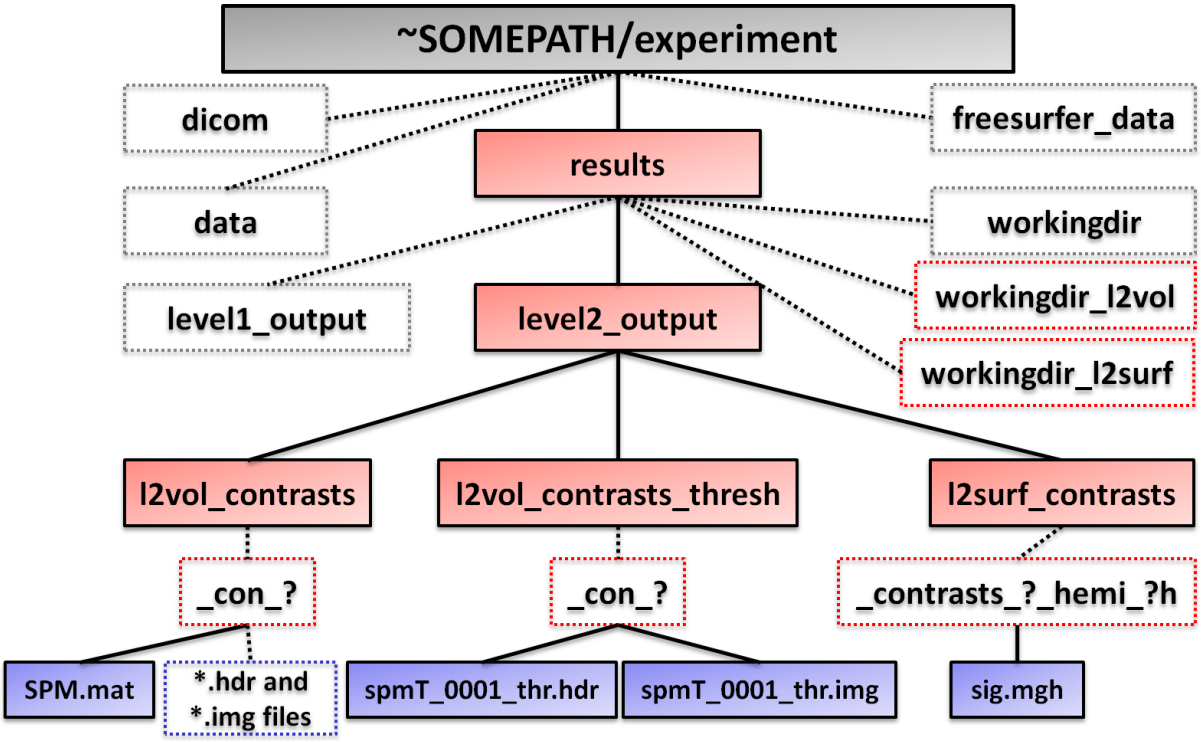


This detailed graph of the `flat` version shows the `l2surfflow`:



6.7 Resulting Folder Structure

After we've run the **second level analysis pipeline** on the volume and the surface our folder structure should look like this:



Additionally to the `level1_output` folder in `~SOMEPATH/experiment/results/` we now have the new folders:

- **workingdir_l2vol** folder that contains all the data that gets created from the `l2volflow` pipeline. As with all working directories, it is **highly recommended** to delete this folder as soon as possible.
- **workingdir_l2surf** folder that contains all the data that gets created from the `l2surfflow` pipeline. As with all working directories, it is **highly recommended** to delete this folder as soon as possible.
- **level2_output** folder which is the datasink of the **second level pipeline**. It contains:
  - **l2vol_contrasts** folder with the estimated volume contrasts and the SPM.mat file for each contrast
  - **l2vol_contrasts_thresh** folder with a thresholded estimated volume contrasts file for each contrast
  - **l2surf_contrasts** folder with the estimated surface contrast file for each hemisphere for each contrast

CHAPTER

SEVEN

HOW TO EXTRACT REGIONS OF INTEREST (ROIS)

In this part we will learn how to extract statistical data from a specified region or also called a **region of interest (ROI)**.

7.1 Anatomical and Functional ROIs

Anatomical and functional ROIs differ mostly by the way how they define the region of interest.

7.1.1 Anatomical ROI

The region of an anatomical ROI is as the name implies defined by the anatomical structure of the brain (e.g. Thalamus, Putamen, Ventricle, Amygdala etc.). Such definitions of anatomical regions and there segmentations is stored in so called **atlas**. A well known atlas which is used by **Nipype** by default is the FreeSurfer Color Table.

This color table subdivides the brain in over 1000 different anatomical regions. The **FreeSurfer Color Table** enables to differentiate between gray and white matter areas. The table is divided into different sections. For the extraction of anatomical ROIs we are interested in the **original Segmentation** and in the **2009 Segmentation**.

Note: Note that this separation between original and 2009 version of segmentation is defined by me and how I would subdivide the color table and by no means represent the structure that FreeSurfer intended.

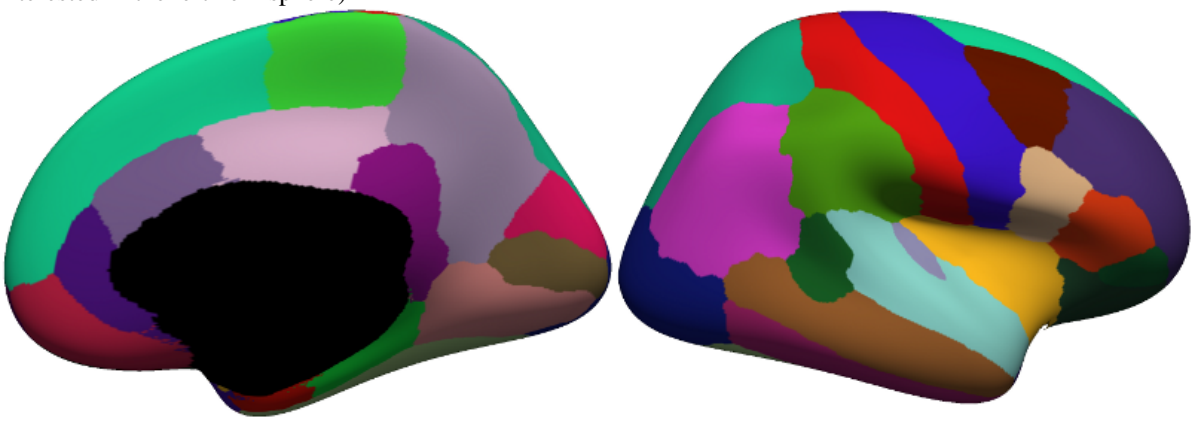
The following list shows a part of the **FreeSurfer Color Table**. Import for us is the segmentation **id** and segmentation **name**, shown in the first two columns.

```
#No. Label Name:                          R   G   B   A
[...]
9   Left-Thalamus                         0   118 14  0
10  Left-Thalamus-Proper                  0   118 14  0
11  Left-Caudate                          122 186 220 0
12  Left-Putamen                          236 13  176 0
13  Left-Pallidum                         12  48  255 0
14  3rd-Ventricle                         204 182 142 0
15  4th-Ventricle                         42  204 164 0
16  Brain-Stem                            119 159 176 0
17  Left-Hippocampus                      220 216 20  0
18  Left-Amygdala                         103 255 255 0
19  Left-Insula                           80  196 98  0
20  Left-Operculum                        60  58  210 0
[...]
48  Right-Thalamus                        0   118 14  0
49  Right-Thalamus-Proper                 0   118 14  0
50  Right-Caudate                         122 186 220 0
51  Right-Putamen                         236 13  176 0
```

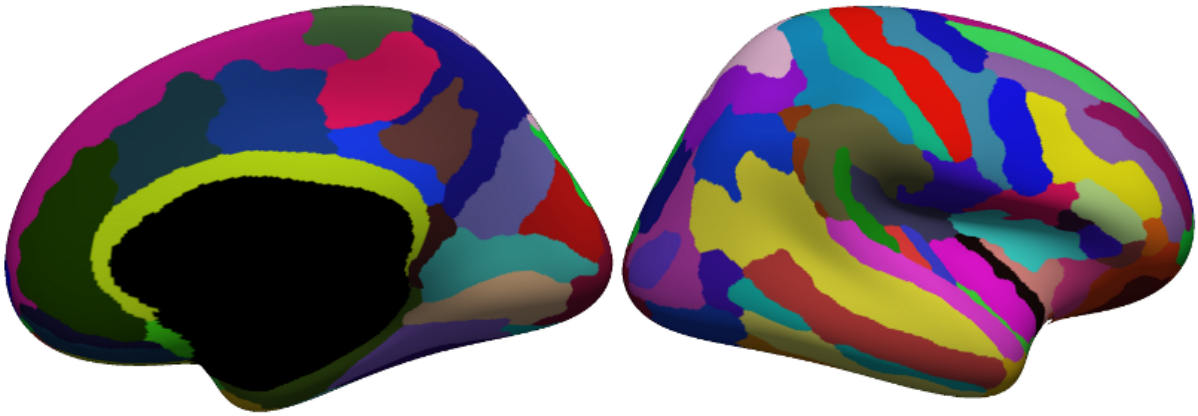
```
52  Right-Pallidum                          13  48  255 0
53  Right-Hippocampus                       220 216 20  0
54  Right-Amygdala                          103 255 255 0
55  Right-Insula                            80  196 98  0
56  Right-Operculum                         60  58  210 0
[...]
```

The ids from *1 to 999* can be found in the original as well as in the 2009 version of segmentation. The ids *1000 to 8014* can only be found in the **original version** and the ids *11100 to 14175* can only be found in the **2009 version** of segmentation.

Visualization of original Segmentation: The following picture shows you the labels of the original segmentation. The annotation of which color belongs to which region is stored in `~SOMEPATH/experiment/freesurfer_data/fsaverage/label/lh.aparc.annot` (if you're interested in the left hemisphere)



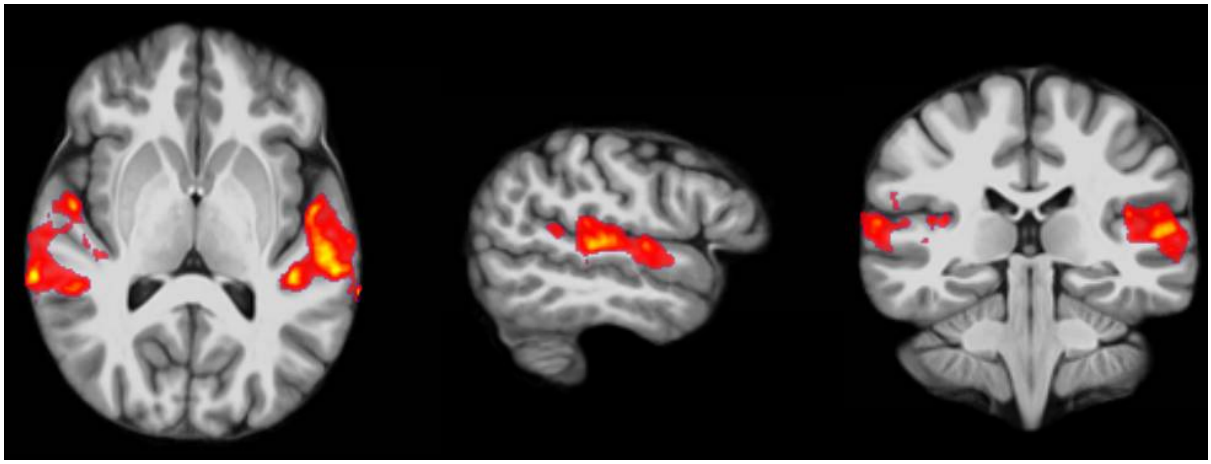
Visualization of 2009 Segmentation: The following picture shows you the labels of the 2009 segmentation. The annotation of which color belongs to which region is stored in `~SOMEPATH/experiment/freesurfer_data/fsaverage/label/rh.aparc.a2009s.annot` (if you're interested in the right hemisphere)



Hint: If you want to explore those regions by ourself I recommend to open a **FreeSurfer** capable viewing tool and overlay the annotation file from the `fsaverage/label` folder.

7.1.2 Functional ROI

The most important difference to the extraction of an anatomical ROI is that the functional region of interest isn't predefined by some atlas but is solely defined by a point in "brain"-space that you are interested in. You're region of interest is most likely a point of strong or interesting activation in your functional data, hence its name. The following is a visualization of a thresholded contrast from our previous second level analysis. The contrast has a cluster threshold of 0.001 and a fdr-p-voxel threshold of 0.05.



The specification of our region of interest can now be done in different ways. Some may want to look at a cubic region with the point of interest as its center. Others want to extract a spherical region around a point of interest. It would also be possible to extract exact exactly all the voxel you can see in the picture above.

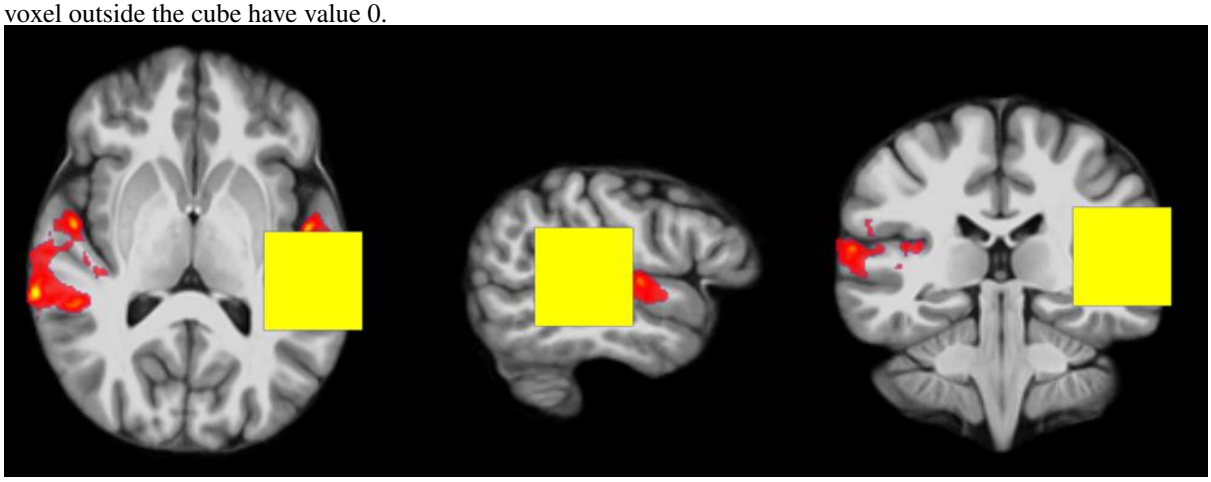
In this example we will extract the data around a spherical ROI with the point [179, 129, 107] in MNI-space as its center and a radius of 20 voxel from the thresholded contrast above. The procedure to do this is quite simple:

1. We define a point of interest and create a sphere around it. This step has to be divided into the following substeps to be achieved:
   (a) Create a cube around your point of interest so that the region you want to be cover is covered
   (b) Transform the cube into a sphere by smoothing it
2. We create a ROI which is the intersection of this spherical ROI and the thresholded activation from the second level analysis
3. We extract the data from this intersection.

Let's take a closer look at those steps.

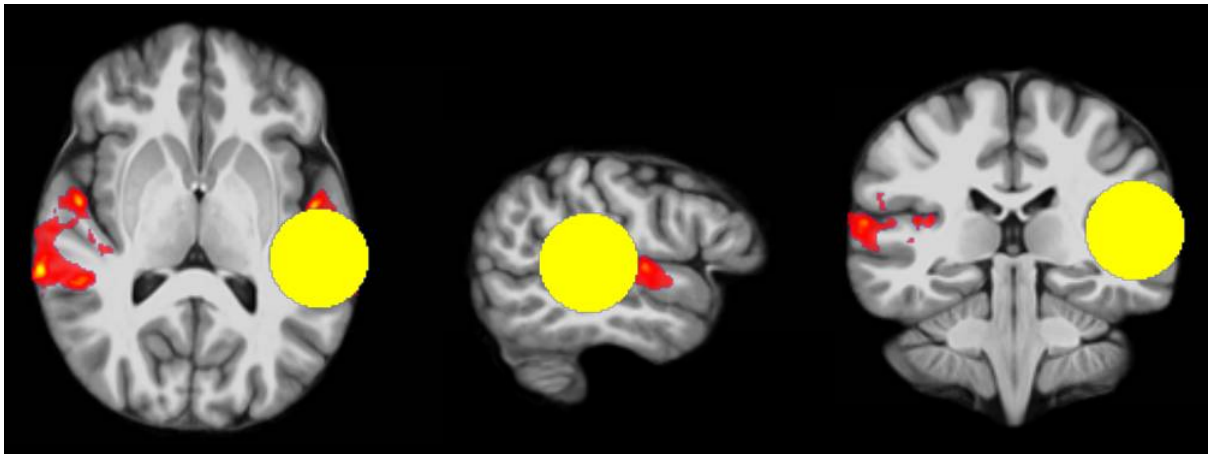
Step 1a: Create a cube around your point of interest

To create this cube around the point of interest some not so obvious steps have to be done. **First**, take a contrast with the dimensions of your normbrain, multiply all voxel by 0 and add the value 1 to each. **Second**, define a cubic area around your point of interest by specifying the lowest corners of the cube and the length of its sides. In this example this is 20. This will left you with a cubic region where all voxel inside the cube have value 1 and all voxel outside the cube have value 0.



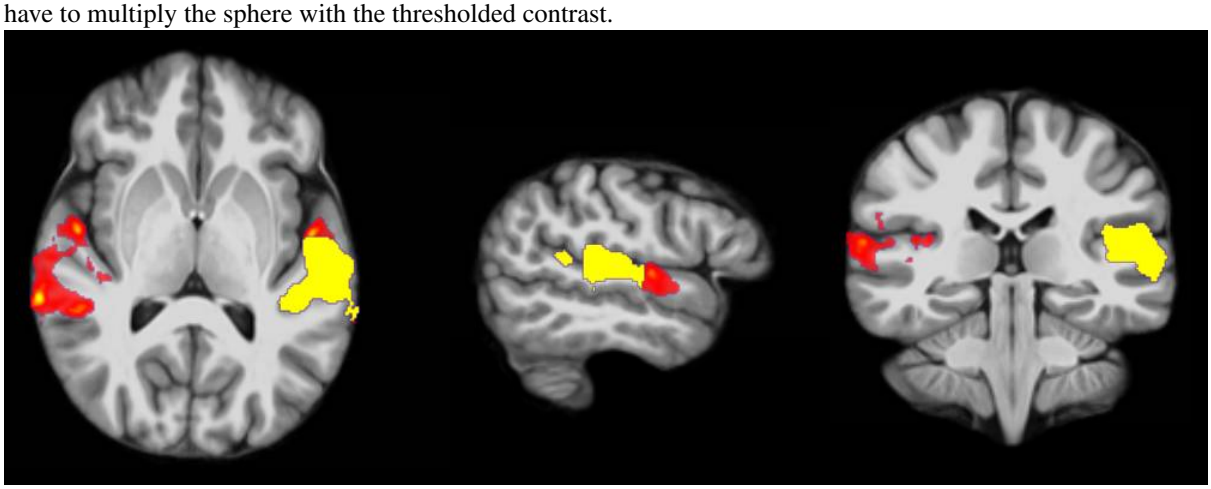
Step 1b: Smooth the cube to a sphere

To do this, take the cube and smooth it to a sphere with radius 20. If you're using `fsl.ImageMaths()` you have to first threshold the sphere with 0.5 and binarize it afterwards. Now you have sphere with the same diameter as the side of the cube was.



Step 2: Create the intersection between the thresholded contrast and the sphere

To create the ROI that is the intersection of the active voxels from the thresholded contrast and the sphere we have have to multiply the sphere with the thresholded contrast.



Step 3: Extract the data from this intersection

By feeding this self defined ROI into `fs.SegStats()` we can extract the values of this region.

7.2 Anatomical ROI Pipeline

Let's now begin with the creation of an **anatomical ROI pipeline**.

7.2.1 Import modules

```
import os                                    # system functions
import nipype.interfaces.freesurfer as fs    # freesurfer
import nipype.interfaces.io as nio           # i/o routines
import nipype.interfaces.utility as util     # utility
import nipype.pipeline.engine as pe          # pypeline engine
```

7.2.2 Define experiment specific parameters

```
#to better access the parent folder of the experiment
experiment_dir = '~SOMEPATH/experiment'

# Tell freesurfer what subjects directory to use
```

```
subjects_dir = experiment_dir + '/freesurfer_data'
fs.FSCommand.set_default_subjects_dir(subjects_dir)

#dirnames for anatomical ROI pipeline
aROIOutput = 'aROI_output'      #name of aROI datasink
l1contrastDir = 'level1_output' #name of first level datasink

#list of subjectnames
subjects = ['subject1', 'subject2', 'subject3']

#list of contrastnumbers the pipeline should consider
contrasts = ['01','02','03','04','05']

#name of the first session from the first level pipeline
nameOfFirstSession = 'func1'
```

Note: The **name of the first session** is necessary, because the bbregister file from the first level pipeline contains the name of the first session in its name. E.g. if the first condition is `func1`, than the name of the bbregister file for the second subject is `meanafunc1_bbreg_subject2.dat`.

7.2.3 Define aROI specific parameters

As mentioned above we are using the FreeSurfer Color Table to define the anatomical regions. Let's assume that we want to extract the following regions:

- id `11` ; `Left-Caudate` ; from **both** segmentations
- id `50` ; `Right-Caudate` ; from **both** segmentations
- id `12` ; `Left-Putamen` ; from **both** segmentations
- id `51` ; `Right-Putamen` ; from **both** segmentations
- id `1007` ; `ctx-lh-fusiform` ; from the **original** segmentation
- id `2007` ; `ctx-rh-fusiform` ; from the **original** segmentation
- id `1022` ; `ctx-lh-postcentral` ; from the **original** segmentation
- id `2022` ; `ctx-rh-postcentral` ; from the **original** segmentation
- id `11134` ; `ctx_lh_G_temp_sup-Lateral` ; from the **2009** segmentation
- id `12134` ; `ctx_rh_G_temp_sup-Lateral` ; from the **2009** segmentation

```
#Specification of the regions from the original and
#the 2009 segmentation version of the FreeSurfer Color Table
ROIregionsorig = ['11','50','12','51','1007','2007','1022','2022']
ROIregions2009 = ['11134','12134']
```

Note: There is no harm by defining a segmentation id in the wrong segmentation version, but if you specify an id between 1000 to 9999 in the `ROIregions2009` list, you just won't extract any other value than 0 from your data.

7.2.4 Define nodes

```
#Node: IdentityInterface - to iterate over subjects and contrasts
inputnode = pe.Node(interface=util.IdentityInterface(fields=['subject_id','contrast_id']),
                    name='inputnode')
inputnode.iterables = [('subject_id', subjects),
                       ('contrast_id', contrasts)]
```

As always when we define the `DataGrabber` node, it is important to be aware of the structure and the files that we want to grab. In this case we want to grab the subject specific contrasts and bbregister file from the first level pipeline.

In our version the **second contrast** for the **third subject** can be found at: `~SOMEPATH/experiment/results/level1_output/vol_contrast/_subject_id_subject3/con0002.img` and the **bbregister file** for the **first subject** can be found at: `~SOMEPATH/experiment/results/level1_output/bbregister/_subject_id_subject1/meanafunc1_bbreg_subject1.dat`. Knowing this, we can build our `datagrabber` node as follows:

```
#Node: DataGrabber - to grab the input data
datasource = pe.Node(interface=nio.DataGrabber(infields=['subject_id','contrast_id'],
                                               outfields=['contrast','bb_id']),
                     name = 'datasource')
datasource.inputs.base_directory = experiment_dir + '/results/' + l1contrastDir
datasource.inputs.template = '%s/_subject_id_%s/%s%s%s'

info = dict(contrast = [['vol_contrasts','subject_id','con_00','contrast_id','.img']],
            bb_id = [['bbregister','subject_id','meana'+nameOfFirstSession+'_bbreg_',
                      'subject_id','.dat']])
datasource.inputs.template_args = info
```

Let's now continue with the implementation of the other nodes that we need for our **anatomical ROI pipeline**.

```
#Node: FreeSurferSource - to grab FreeSurfer files from the recon-all process
fssource = pe.Node(interface=nio.FreeSurferSource(),name='fssource')
fssource.inputs.subjects_dir = subjects_dir

#Node: MRIConvert - to convert files from FreeSurfer format into nifti format
MRIconversion = pe.Node(interface=fs.MRIConvert(),name='MRIconversion')
MRIconversion.inputs.out_type = 'nii'

#Node: ApplyVolTransform - to transform contrasts into anatomical space
#                          creates 'con_*.anat.bb.mgh' files
transformation = pe.Node(interface=fs.ApplyVolTransform(),name='transformation')
transformation.inputs.fs_target = True
transformation.inputs.interp = 'nearest'

#Node: SegStatsorig - to extract specified regions of the original segmentation
segmentationorig = pe.Node(interface=fs.SegStats(),name='segmentationorig')
segmentationorig.inputs.segment_id = ROIregionsorig

#Node: SegStats2009 - to extract specified regions of the 2009 segmentation
segmentation2009 = pe.Node(interface=fs.SegStats(),name='segmentation2009')
segmentation2009.inputs.segment_id = ROIregions2009

#Node: Datasink - Creates a datasink node to store important outputs
datasink = pe.Node(interface=nio.DataSink(), name="datasink")
datasink.inputs.base_directory = experiment_dir + '/results'
datasink.inputs.container = aROIOutput
```

Note: We implement two `SegStats` nodes because we have two lists of segmentation, the `original` and the `2009` one. It might be possible two feed those values into one node with using iterables, but I prefer this way.

If you don't specify a value for `segment_id` the pipeline will extract all regions of your atlas. This is not bad but can take a long time.

7.2.5 Definition of anatomical ROI workflow

```
#Initiation of the ROI extraction workflow
aROIflow = pe.Workflow(name='aROIflow')
aROIflow.base_dir = experiment_dir + '/results/workingdir_aROI'
```

Before we can start with the integration of the nodes into `aROIflow` we have to be aware about something. If we would try to simply connect `fssource` with `segmentationorig` like this...

```
...(fssource, segmentationorig,[('aparc_aseg','segmentation_file')]),
```

... we would get the following error:

```
Node: segmentationorig
input: segmentation_file
TraitError: The 'segmentation_file' trait of a SegStatsInputSpec instance must be an
existing file name, but a value of ['~SOMEPATH/experimentfreesurfer_data/
subject1/mri/aparc+aseg.mgz', '~SOMEPATH/experiment/freesurfer_data/subject1/mri
/aparc.a2009s+aseg.mgz'] <type 'list'> was specified.
```

This means that the output of the `fssource` node has the value: `['~SOMEPATH/experiment/freesurfer_data/subject1/mri/aparc+aseg.mgz', '~SOMEPATH/experiment/freesurfer_data/subject1/mri/aparc.a2009s+aseg.mgz']`. That means we have to guide the output of this node so that `segmentationorig` receives the path to `aparc+aseg.mgz` and that `segmentation2009` receives the path to `aparc.a2009s+aseg.mgz`.

This can be achieved with a simple function:

```
def getSegVersion(in_file, version):
    if version == 0:
       return in_file[0]
    else:
       return in_file[1]
```

Now we are ready to build our **anatomical ROI pipeline**.

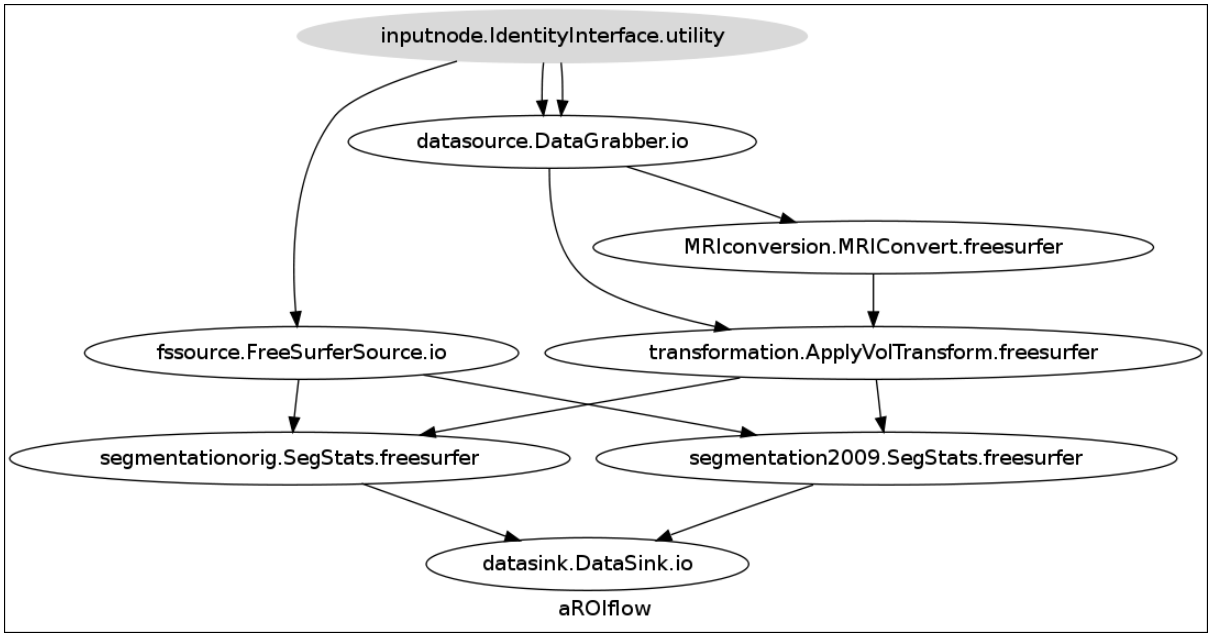
```
#Connect up all components
aROIflow.connect([(inputnode, datasource,[('subject_id','subject_id'),
                                          ('contrast_id','contrast_id'),
                                          ]),
                  (inputnode, fssource,[('subject_id','subject_id')]),
                  (fssource, segmentationorig,[(('aparc_aseg',getSegVersion,0),
                                               'segmentation_file')]),
                  (fssource, segmentation2009,[(('aparc_aseg',getSegVersion,1),
                                               'segmentation_file')]),
                  (datasource, MRIconversion,[('contrast','in_file')]),
                  (MRIconversion, transformation,[('out_file','source_file')]),
                  (datasource, transformation,[('bb_id','reg_file')]),
                  (transformation, segmentationorig,[('transformed_file',
                                                      'in_file')]),
                  (transformation, segmentation2009,[('transformed_file',
                                                      'in_file')]),
                  (segmentationorig, datasink,[('summary_file', 'segstatorig')]),
                  (segmentation2009, datasink,[('summary_file', 'segstat2009')]),
                  ])
```

7.2.6 Run pipeline and generate graph

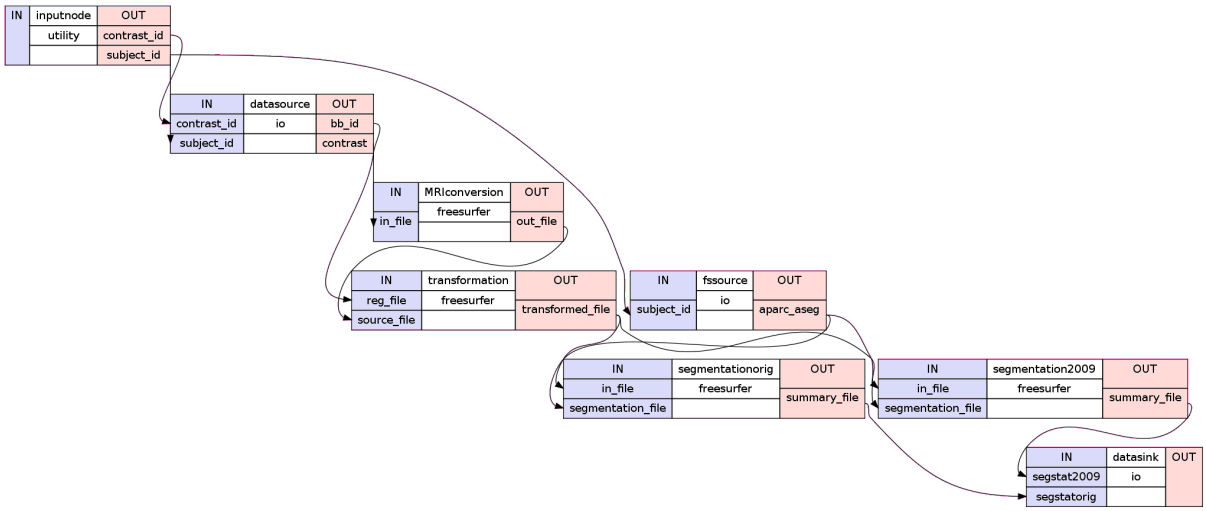
```
aROIflow.write_graph(graph2use='flat')
aROIflow.run(plugin='MultiProc', plugin_args={'n_procs' : 2})
```

7.2.7 Visualization of the anatomical ROI pipeline

This graph of the `hierarchical` version shows the `aROIflow`:



This detailed graph of the `flat` version shows the `aROIflow`:



7.2.8 Summarizing the output in a cvs-file

This part is optional and shows only one of many ways how you can summarize the output of `aROIflow` into a common cvs-file. In essence it does the following steps for each contrast:

- grab contrast and subject specific `summary.stats` file
- extract values from this file and store it into a list
- add the list of each subject into a bigger list
- store this big list into a csv-file

```
#iterate over contrasts and create a cvs-file for each
for contrast in contrasts:

    #creates a list with an empty entry for each segmentation id
    output = []
    for i in range(15000):
        output.append([i,None])

    #to keep track of added subjects
```

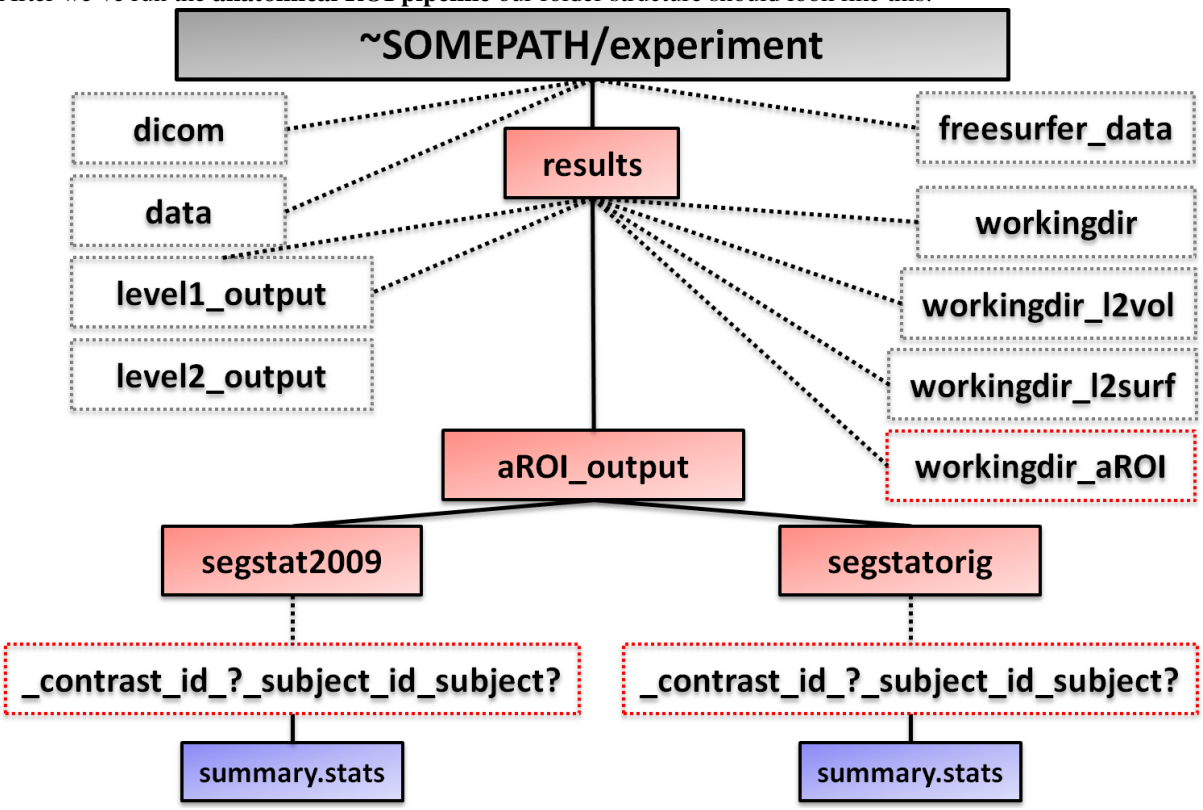
```
    subjectNumber = 1

    #iterate over subjects and entering values into output
    for subject in subjects:

        #specify path to aROI datasink for each variation of segmentation
        path2aROIOut = experiment_dir+'/results/'+aROIOutput
        path2Sumfile = '_contrast_id_'+contrast+'_subject_id_'+subject+'/summary.stats'
        statsFileorig = path2aROIOut+'/segstatorig/'+path2Sumfile
        statsFile2009 = path2aROIOut+'/segstat2009/'+path2Sumfile

        #extract the data from the output summary files
        dataFile = open(statsFileorig, 'r')
        dataorig = dataFile.readlines()
        dataFile.close()
        dataFile = open(statsFile2009, 'r')
        data2009 = dataFile.readlines()
        dataFile.close()

        #function to check where the data starts
        def findStartOfData(datafile):
            for line in range(100):
                if datafile[line][0] != '#':
                   return line

        #get data and store it in tempresult
        tempresult = []

        for line in range(len(dataorig)):
            if line < findStartOfData(dataorig):
                pass
            else:
                temp = dataorig[line].strip('\n').split()
                tempresult.append([int(temp[1]),temp[4],float(temp[5])])

        for line in range(len(data2009)):
            if line < findStartOfData(data2009):
                pass
            else:
                temp = data2009[line].strip('\n').split()
                tempresult.append([int(temp[1]),temp[4],float(temp[5])])

        tempresult.sort()

        result = []

        for line in range(len(tempresult)):
            #pass if region has already been added
            if line > 0 and tempresult[line] == tempresult[line-1]:
                pass
            else:
                result.append(tempresult[line])

        for ROI in result:
            #if id wasn't extracted before, adds name of id to row
            if output[ROI[0]][1] == None:
               output[ROI[0]][1] = ROI[1]
            #adds value of id into row
            output[ROI[0]].append(ROI[2])

        #if no value for an id was entered for a subject
        #   the value 0.0 gets added
        for ROI in output:
```

```
            if len(ROI) < subjectNumber+2:
                ROI.append(0.0)

        subjectNumber += 1

    #adds labels to the first row of the output
    output.insert(0,['SegId','StructName'])
    output[0].extend(subjects)

    #adds segment if it was extracted
    output = [ROI for ROI in output if ROI[1] != None]

    #define name of the output csv-file
    summaryFileName = 'aROI_'+contrast+'.csv'

    #store output into a cvs-file
    f = open(path2aROIOut+'/'+summaryFileName,'wb')
    import csv
    outputFile = csv.writer(f)
    for line in output:
        outputFile.writerow(line)
    f.close()
```

Hint: The code for this anatomical ROI pipeline can be found here: aROIpipeline.py

7.2.9 Folder structure after `aROIflow`

After we've run the **anatomical ROI pipeline** our folder structure should look like this:



Additionally to the `level2_output` folder in `~SOMEPATH/experiment/results/` we now have the new folders:

- **workingdir_aROI** folder that contains all the data that gets created from the `aROIflow` pipeline. As with all working directories, it is **highly recommended** to delete this folder as soon as possible.

- **aROI_output** folder which is the datasink of the **anatomical ROI pipeline**. It contains:
  - **segstatorig** folder with the `summary.stats` file from the original segmentation for each contrast and subject
  - **segstat2009** folder with the `summary.stats` file from the 2009 segmentation for each contrast and subject

Note: If you run the additional python code that is at the end of the script, the summarization files that get created will be stored in `~SOMEPATH/experiment/results/aROI_output`.

7.3 Functional ROI Pipeline

Before we start with the creation of our **functional ROI pipeline**, let's have a closer look at step 1a, 1b and 2 from the introduction part above. To do those steps we use `fslmaths` from the FSL interface which we will be implementing in Nipype with the `fsl.ImageMaths()` interface. Sometimes it is helpfull or necessary to know what the final command will be before creating the nodes.

Step 1a will be implemented by a node called `cubemask`. This node will create the following FSL command:

```
fslmaths ~SOMEPATH/experiment/results/level1_output/normcons/subject2/
con_0001_ants.nii -mul 0 -add 1 -roi 159 40 109 40 87 40 0 1 ~SOMEPATH/experiment/
results/workingdir_fROI/fROIflow/cubemask/con_0001_ants_maths.nii.gz -odt float
```

Step 1b will be implemented by a node called `spheremask`. This node will create the following FSL command:

```
fslmaths ~SOMEPATH/experiment/results/workingdir_fROI/fROIflow/cubemask/
con_0001_ants_maths.nii.gz -kernel sphere 20 -fmean -thr 0.5 -bin ~SOMEPATH/experiment/
results/workingdir_fROI/fROIflow/spheremask/con_0001_ants_maths_maths.nii.gz -odt float
```

Step 2 will be implemented by a node called `tmapmask`. This node will create the following FSL command:

```
fslmaths ~SOMEPATH/experiment/results/workingdir_fROI/fROIflow/spheremask/
con_0001_ants_maths_maths.nii.gz -mul ~SOMEPATH/experiment/results/level2_output/
l2vol_contrasts_thresh/_con_4/spmT_0001_thr.hdr -bin ~SOMEPATH/experiment/
results/workingdir_fROI/fROIflow/_contrast_id_4_subject_id_subject2/tmapmask/
con_0001_ants_maths_maths_maths.nii.gz -odt float
```

Note: If you are interested in what commandline an interface like `fsl.ImageMaths()` actually will be executing use the command `nodename.cmdline` (e.g. `cubemask.cmdline` if `cubemask = fsl.ImageMaths()`). But for this to work `cubemask` mustn't be defined as a node like `pe.Node(interface=fsl.ImageMaths(),name="cubemask")`.

More informations about the different arguments of `fslmaths` can be found here. To be on the safe side, I additionally copied it's content into fslmaths.txt.

Important: This document wasn't created by me, I just stumbled upon it. But it is the best and most detailed one I've found.

How those FSL commands can be achieved by using Nipype can be found in the code later on. Let's now start building our **functional ROI pipeline**.

7.3.1 Import modules

```
1 import os                                    # system functions
2 import nipype.interfaces.freesurfer as fs    # freesurfer
3 import nipype.interfaces.io as nio           # i/o routines
4 import nipype.interfaces.utility as util     # utility
5 import nipype.pipeline.engine as pe          # pypeline engine
6 import nipype.interfaces.fsl as fsl          # fsl module
```

7.3.2 Define experiment specific parameters

```
#to better access the parent folder of the experiment
experiment_dir = '~SOMEPATH/experiment'

#dirnames for functional ROI and of level1 datasink
fROIOutput = 'fROI_output'      #name of fROI datasink
l1contrastDir = 'level1_output' #name of first level datasink

#list of subjectnames
subjects = ['subject1','subject2','subject3']

#list of contrastnumbers the pipeline should consider
contrasts = [1,2,3,4,5]
```

7.3.3 Define fROI specific parameters

```
#define the coordination of the point of interest
centerOfROI = [179,129,107]

#define the radius of the sphere of interest
radius = 20

#calculates the beginning corner of the cubic ROI
corner = [centerOfROI[0]-radius,
          centerOfROI[1]-radius,
          centerOfROI[2]-radius]
```

7.3.4 Definition of Nodes

```
#Node: IdentityInterface - to iterate over subjects and contrasts
inputnode = pe.Node(interface=util.IdentityInterface(fields=['subject_id','contrast_id']),
                    name='inputnode')
inputnode.iterables = [('subject_id', subjects),
                       ('contrast_id', contrasts)]

#Node: DataGrabber - To grab the input data
datasource = pe.Node(interface=nio.DataGrabber(infields=['subject_id','contrast_id'],
                                               outfields=['contrast']),
                     name = 'datasource')
datasource.inputs.base_directory = experiment_dir + '/results/' + l1contrastDir
datasource.inputs.template = 'norm%s/%s/%s_%04d_ants.nii'
info = dict(contrast = [['cons','subject_id','con','contrast_id']])
datasource.inputs.template_args = info

#Node: ImageMaths - to create the cubic ROI with value 1
cubemask = pe.Node(interface=fsl.ImageMaths(),name="cubemask")
cubeValues = (corner[0],radius*2,corner[1],radius*2,corner[2],radius*2)
cubemask.inputs.op_string = '-mul 0 -add 1 -roi %d %d %d %d %d %d 0 1'%cubeValues
cubemask.inputs.out_data_type = 'float'
pathValues = (subjects[0],contrasts[0])
cubemask.inputs.in_file = '~SOMEPATH/experiment/normcons/%s/con_%04d_ants.nii'%pathValues

#Node: ImageMaths - to transform the cubic ROI to a spherical ROI
spheremask = pe.Node(interface=fsl.ImageMaths(),name="spheremask")
spheremask.inputs.op_string = '-kernel sphere %d -fmean -thr 0.5 -bin'%radius
spheremask.inputs.out_data_type = 'float'

#Node: ImageMaths - to mask the spherical ROI with a subject specific T-map
```

```
tmapmask = pe.Node(interface=fsl.ImageMaths(),name="tmapmask")
tmapmask.inputs.out_data_type = 'float'

#function to add the thresholded group T-map to op_string
def groupTMapPath(contrast_id):
    experiment_dir = '~SOMEPATH/experiment'
    op_string = '-mul %s/results/level2_output/l2vol_contrasts_thresh/_con_%d/spmT_0001_thr.hdr -
    return op_string%(experiment_dir,contrast_id)

#Node: SegStats - to extract the statistic from a given segmentation
segstat = pe.Node(interface=fs.SegStats(),name='segstat')

#Node: Datasink - Create a datasink node to store important outputs
datasink = pe.Node(interface=nio.DataSink(), name="datasink")
datasink.inputs.base_directory = experiment_dir + '/results'
datasink.inputs.container = fROIOutput
```

7.3.5 Definition of functional ROI workflow

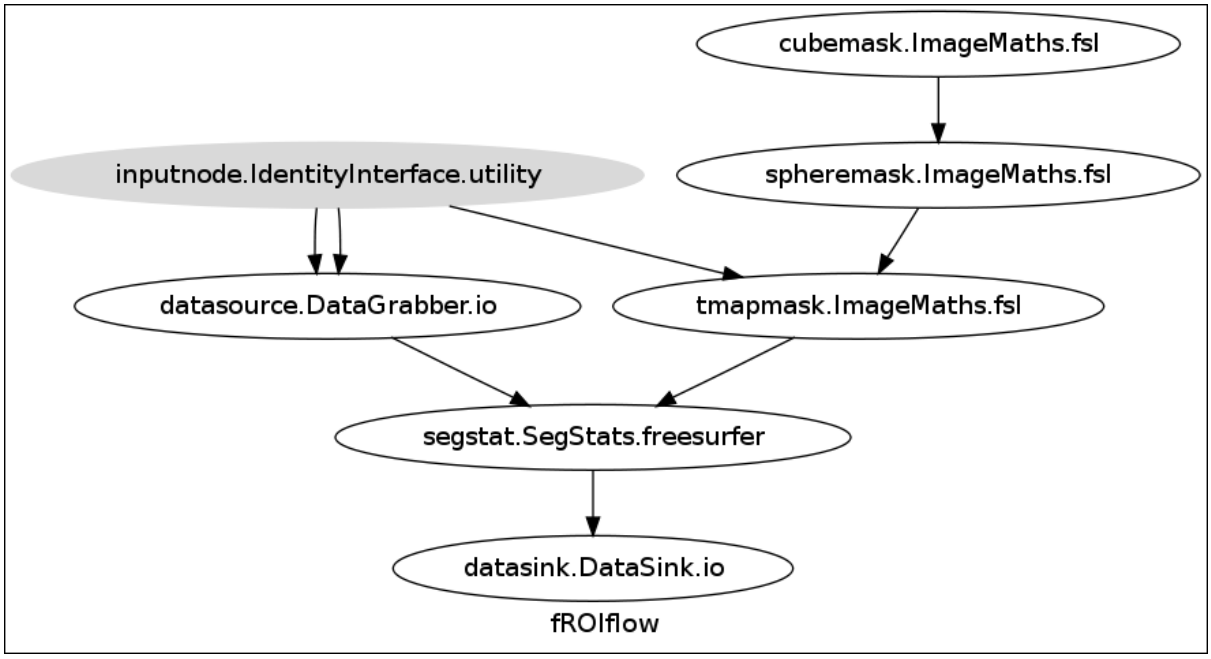
```
#Initiation of the fROI extraction workflow
fROIflow = pe.Workflow(name='fROIflow')
fROIflow.base_dir = experiment_dir + '/results/workingdir_fROI'

#Connect up all components
fROIflow.connect([(cubemask, spheremask,[('out_file','in_file')]),
                  (spheremask, tmapmask,[('out_file','in_file')]),
                  (inputnode, tmapmask,[(('contrast_id',groupTMapPath),
                                         'op_string')
                                        ]),
                  (inputnode, datasource,[('subject_id','subject_id'),
                                          ('contrast_id','contrast_id')
                                          ]),
                  (tmapmask, segstat,[('out_file','segmentation_file')]),
                  (datasource, segstat,[('contrast','in_file')]),
                  (segstat, datasink,[('summary_file','@statistic')]),
                  ])
```

7.3.6 Run the pipeline and generate the graph

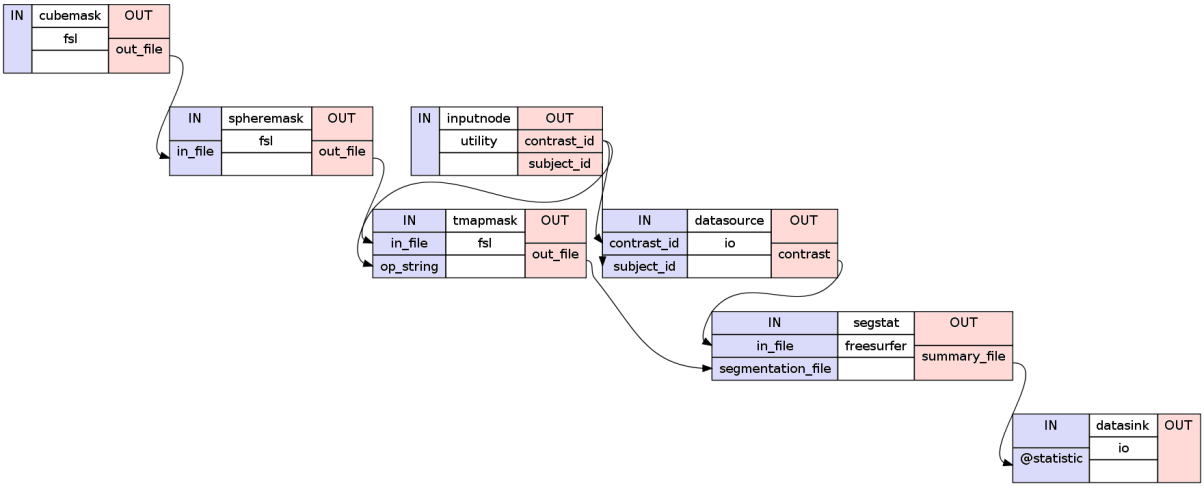
```
fROIflow.write_graph(graph2use='flat')
fROIflow.run(plugin='MultiProc', plugin_args={'n_procs' : 4})
```

7.3.7 Visualization of the functional ROI pipeline

This graph of the `hierarchical` version shows the `fROIflow`:



This detailed graph of the `flat` version shows the `fROIflow`:



7.3.8 Summarizing the output in a cvs-file

This part is optional and shows only one of many ways how you can summarize the output of `fROIflow` into a common cvs-file. In essence it does the following steps:

- grab contrast and subject specific summary.stats file
- extract values from this file and store it into a list
- add the list of each subject into a bigger list
- store this big list into a csv-file

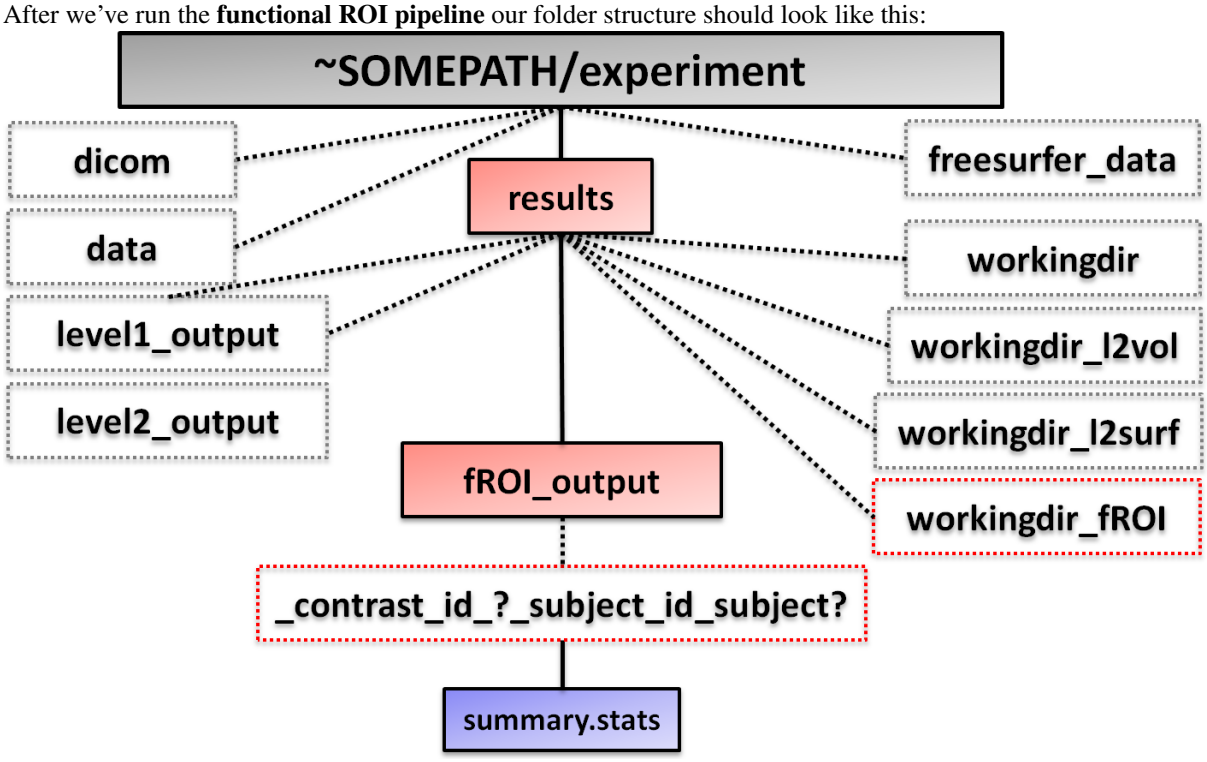
```
#creates the big list and its header
output = []
output.append(['coordinations:',centerOfROI,'radius:',radius])

#iterate over contrasts
for contrast in contrasts:

    #creates header for each contrast
    contrast = str(contrast)
```

```
    output.append(['contrast:',contrast])

    #iterate over subjects
    for subject in subjects:

        #specify path to fROI datasink for each variation of segmentation
        path2fROIOut = experiment_dir+'/results/'+fROIOutput + '/'
        path2Sumfile = '_contrast_id_'+contrast+'_subject_id_'+subject
        statFile = path2fROIOut + path2Sumfile + '/summary.stats'

        #extract the data from the output summary files
        dataFile = open(statFile, 'r')
        data = dataFile.readlines()
        dataFile.close()

        #add value of functional region
        output.append([subject,data[-1].split()[5]])

    #add an empty line at the end of a contrast summary
    output.append([])

#store output into a cvs-file
f = open(path2fROIOut+'/fROI_spherical'+str(centerOfROI)+'_%s_result.csv'%radius,'wb')
import csv
outputFile = csv.writer(f)
for line in output:
    outputFile.writerow(line)
f.close()
```

Hint: The code for this functional ROI pipeline can be found here: fROIpipeline.py

7.3.9 Folder structure after `fROIflow`

After we've run the **functional ROI pipeline** our folder structure should look like this:



Additionally to the `level2_output` folder in `~SOMEPATH/experiment/results/` we now have the new folders:

- **workingdir_fROI** folder that contains all the data that gets created from the `fROIflow` pipeline. As with all working directories, it is **highly recommended** to delete this folder as soon as possible.
- **fROI_output** folder which is the datasink of the **functional ROI pipeline**. In this folder you'll find the `summary.stats` file for the defined functional region for each contrast and subject

Note: If you run the additional python code that is at the end of the script, the summarization files that get created will be stored in `~SOMEPATH/experiment/results/fROI_output`.

CHAPTER

EIGHT

IDEAS FOR FUTURE CHAPTERS

As Nipype itself, this user guide isn't finished yet and there's a lot more to come. So far my plans for topics for future chapters are the following:

- How to use **slicer** to create nice **functional pictures**
- How to use **ANTS** to **normalize your subject specific data** from subject space into a common template space
- How to use **ANTS** to **create your own normbrain template**
- How to **track a running workflow** with report
- How to use **nipype predefined workflows** like `nipype.workflows.freesurfer.utils`

As I try to improve this user guide to make the learning curve for Nipype beginners as steep as possible, feedback, criticism, comments, ideas, recommendation etc. are highly appreciated: mnotter@mit.edu

Hint: If you want this user guide as a PDF file you can download it here: Nipype Beginner's Guide

If you want to download all scripts that are used in this user guide you can find them here.