Nipype Beginner's Guide

Release 1.0

Michael Notter

Dec 04, 2017

CONTENTS

CHAPTER

ONE

INTRODUCTION TO NIPYPE

1.1 What is Nipype?

Nipype (http://nipype.readthedocs.io/en/latest/) (Neuroimaging in Python - Pipelines and Interfaces) is an open-source, user-friendly, community-developed software package under the umbrella of NiPy (http://nipy.org/). Nipype allows you to pipeline your neuroimaging workflow in an intuitive way and enables you to use the software packages and algorithms you want to use, regardless their programing language. This is possible because Nipype provides an uniform interface to many existing neuroimaging processing and analysis packages like SPM (http://www.fil.ion.ucl.ac.uk/spm), FreeSurfer (http://surfer.nmr.mgh.harvard.edu/), FSL (http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/), AFNI (https://afni.nimh.nih.gov/afni/), ANTS (http://stnava.github.io/ANTs/), Camino (http://camino.cs.ucl.ac.uk/), MRtrix (http://www.brain.org.au/software/mrtrix/index.html), Slicer (http://slicer.org/), MNE (https://martinos.org/mne/stable/index.html) and many more.

Nipype allows you to easily combine all those heterogeneous software packages whithin a single workflow. This procedure gives you the opportunity to pick the best algorithm there is for the problem at hand and therefore allows you to profit from the advantages of any software package you like.

Nipype is written in Python (https://www.python.org/), an easy to learn and very intuitive programming language. This means that your whole neuroimaging analysis can be easily specified using python scripts. It won't even take as many lines of code as you might fear. Nipype is very straightforward and easy to learn. As you will see, it is quite simple to combine processing steps using different software packages. Steps from previous analyses can be reused effortlessly and new approaches can be applied much faster.

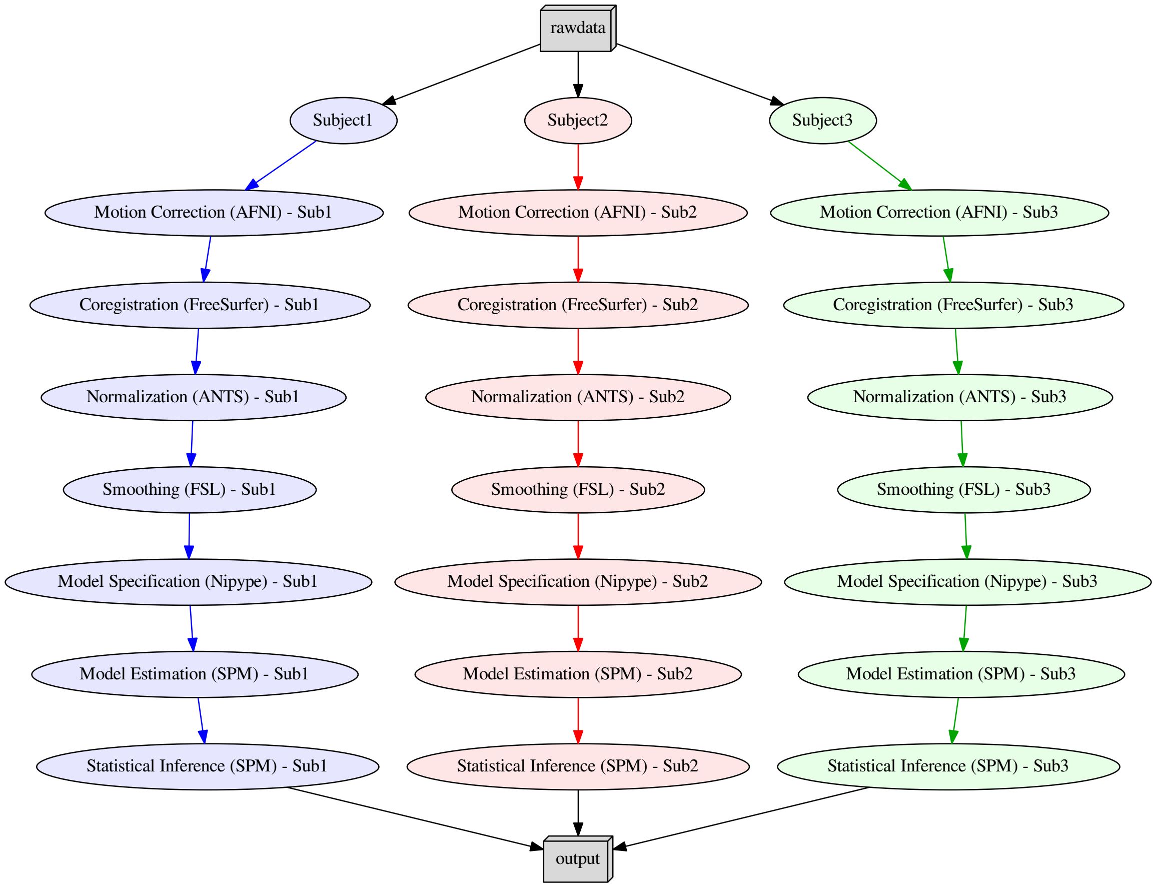
You're still concerned because you want to combine your own **bash**, **MATLAB**, **R** or **Python** scripts with Nipype? No problem! Even the creation of your own interface to your own software solution is straightforward and can be done in a rather short time. Thanks to Github (https://github.com/nipy/nipype), there's also always a community standing behind you.

Nipype provides an environment that encourages interactive exploration of algorithms. It allows you to make your research easily reproducible and lets you share your code with the community.

1.2 A Short Example

Let's assume you want to do an Analysis that uses **AFNI** for the *Motion Correction*, **FreeSurfer** for the *Coregistration*, **ANTS** for the *Normalization*, **FSL** for the *Smoothing*, **Nipype** for the *Model Specification*, **SPM** for the *Model Estimation* and **SPM** for the *Statistical Inference*. Normally this would be a hell of a mess. Switching between multiple scripts in different programming languages with a lot of manual intervention. On top of all that, you want to do your analysis on multiple subjects, preferably as fast as possible, i.e., processing several subjects in parallel. With Nipype, this is no problem! You can do all this and much more.

To illustrate the straightforwardness of an Nipype workflow and show how simply it can be created, look at the following example. This figure shows you a simplification of the analysis *workflow* just outlined.


rawdata
Subject1
Subject2
Subject3
Motion Correction (AFNI) - Sub1
Motion Correction (AFNI) - Sub2
Motion Correction (AFNI) - Sub3
Coregistration (FreeSurfer) - Sub1
Coregistration (FreeSurfer) - Sub2
Coregistration (FreeSurfer) - Sub3
Normalization (ANTS) - Sub1
Normalization (ANTS) - Sub2
Normalization (ANTS) - Sub3
Smoothing (FSL) - Sub1
Smoothing (FSL) - Sub2
Smoothing (FSL) - Sub3
Model Specification (Nipype) - Sub1
Model Specification (Nipype) - Sub2
Model Specification (Nipype) - Sub3
Model Estimation (SPM) - Sub1
Model Estimation (SPM) - Sub2
Model Estimation (SPM) - Sub3
Statistical Inference (SPM) - Sub1
Statistical Inference (SPM) - Sub2
Statistical Inference (SPM) - Sub3
output

The code to create an Nipype workflow that specifies the steps illustrated in the figure above and can run all the steps would look something like this:

```python
#Import modules
import nipype
import nipype.interfaces.afni        as afni
import nipype.interfaces.freesurfer  as fs
import nipype.interfaces.ants        as ants
import nipype.interfaces.fsl         as fsl
import nipype.interfaces.nipy        as nipy
import nipype.interfaces.spm         as spm


#Specify experiment specifc parameters
experiment_dir = '~/experiment_folder'
nameofsubjects = ['subject1','subject2','subject3']

#Where can the raw data be found?
grabber = nipype.DataGrabber()
grabber.inputs.base_directory = experiment_dir + '/data'
grabber.inputs.subject_id = nameofsubjects

#Where should the output data be stored at?
sink = nipype.DataSink()
```

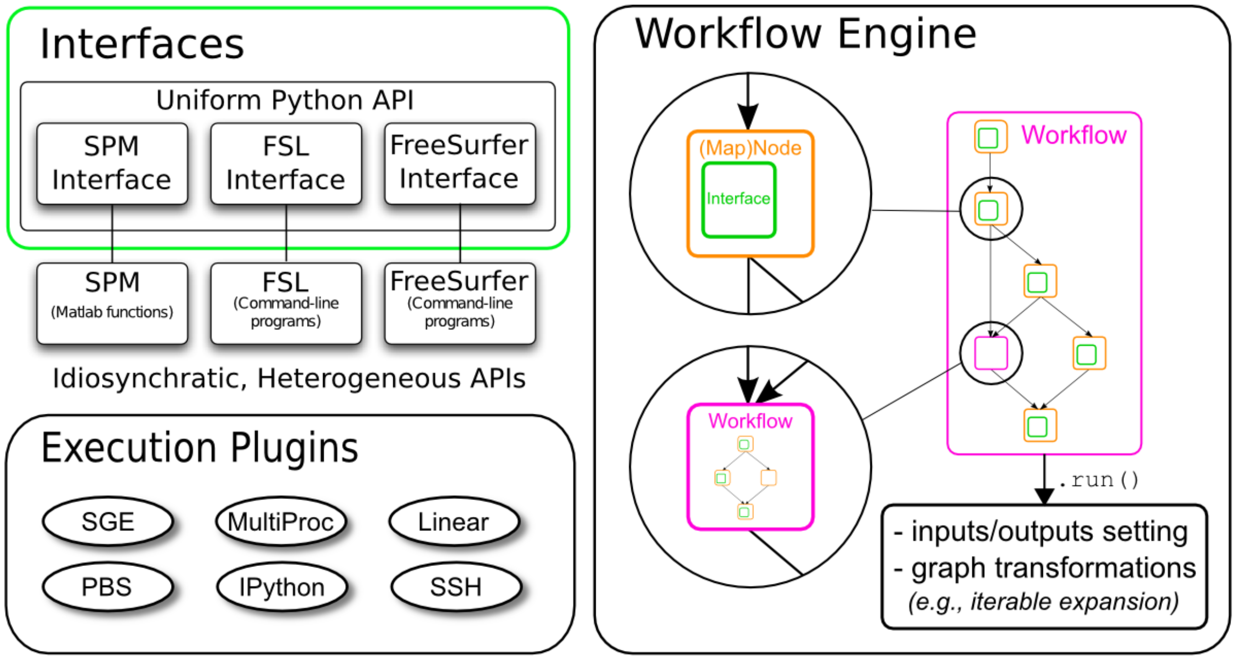
```python
sink.inputs.base_directory = experiment_dir + '/output_folder'


#Create a node for each step of the analysis

#Motion Correction (AFNI)
realign = afni.Retroicor()

#Coregistration (FreeSurfer)
coreg = fs.BBRegister()

#Normalization (ANTS)
normalize = ants.WarpTimeSeriesImageMultiTransform()

#Smoothing (FSL)
smooth = fsl.SUSAN()
smooth.inputs.fwhm = 6.0

#Model Specification (Nipype)
modelspec = nipype.SpecifyModel()
modelspec.inputs.input_units = 'secs'
modelspec.inputs.time_repetition = 2.5
modelspec.inputs.high_pass_filter_cutoff = 128.

#Model Estimation (SPM)
modelest = spm.EstimateModel()

#Contrast Estimation (SPM)
contrastest = spm.EstimateContrast()
cont1 = ['human_faces',  [1 0 0]]
cont2 = ['animal_faces', [0 1 0]]
contrastest.inputs.contrasts = [cont1, cont2]

#Statistical Inference (SPM)
threshold = spm.Threshold()
threshold.inputs.use_fwe_correction = True
threshold.inputs.extent_fdr_p_threshold = 0.05


#Create a workflow to connect all those nodes
analysisflow = nipype.Workflow()

#Connect the nodes to each other
analysisflow.connect([(grabber     ->  realign    ),
                      (realign     ->  coreg      ),
                      (coreg       ->  normalize  ),
                      (normalize   ->  smooth     ),
                      (smooth      ->  modelspec  ),
                      (modelspec   ->  modelest   ),
                      (modelest    ->  contrastest),
                      (contrastest ->  threshold  ),
                      (threshold   ->  sink       )
                      ])

#Run the workflow in parallel
analysisflow.run(mode='parallel')
```

By using *multicore processing*, *SGE*, *PBS*, *Torque*, *HTCondor*, *LSF* or other plugins for parallel execution you will be able to reduce your computation time considerably. This means, that an analysis of 24 subjects where each takes one hour to process would normally take about one day, but it could be done on a single machine with eight processors in under about three hours.

Note: The code above is of course a shortened and simplified version of the real code. But it gives you a good idea of what the code would look like, and how straightforward and readable the programming of a neuroimaging pipeline with Nipype is.

1.3 Nipype's Architecture

Nipype consists of many parts, but the most important ones are **Interfaces**, the **Workflow Engine** and the **Execution Plugins**.



Note: For a deeper understanding of Nipype go either to Nipype's main homepage (http://nipype.readthedocs.io/en/latest/) or read the official paper: Gorgolewski K, Burns CD, Madison C, Clark D, Halchenko YO, Waskom ML, Ghosh SS (2011) **Nipype: a flexible, lightweight and extensible neuroimaging data processing framework in Python.** *Front. Neuroinform. 5:13.* http://dx.doi.org/10.3389/fninf.2011.00013 (http://journal.frontiersin.org/article/10.3389/fninf.2011.00013/abstract)

1.3.1 Interfaces

Interfaces in the context of Nipype are program wrappers that enable Nipype, which runs in Python, to run a program or function in any other programming language. As a result, Python becomes the common denominator of all neuroimaging software packages and allows Nipype to easily connect them to each other. A short tutorial about interfaces can be found on the official homepage (http://nipype.readthedocs.io/en/latest/users/interface_tutorial.html). More practical examples will be given later in this beginner's guide.

For a full list of software interfaces supported by Nipype go here (http://nipype.readthedocs.io/en/latest/documentation.html).

1.3.2 Workflow Engine

The core of Nipype's architecture is the workflow engine. It consists of **Nodes**, **MapNodes** and **Workflows**, which can be interconnected in various ways.

- **Node**: A node provides the information – parameters, filenames, etc. – that is needed by an interface to run the program properly for a particular job, whether as part of a workflow or separately.
- **MapNode**: A Mapnode is quite similar to a Node, but it differs because it takes multiple inputs of a single type to create a single output. For example, it might specify multiple DICOM files to create one NIfTI file.
- **Workflow**: A workflow (also called a pipeline), is a directed acyclic graph (DAG) or forest of graphs whose nodes are of type Node, MapNode or Workflow and whose edges (lines connecting nodes) represent data flow.

Each Node, MapNode or Workflow has (at least) one input field and (at least) one output field. Those fields specify the dataflow into and out of a Node, MapNode or Workflow. MapNodes use fields to specify multiple inputs (basically a list of input items). There they are called *iterfields* because the interface will iterate over the list of input items, and they have to be labeled as such to distinguish them from single-item fields.

A very cool feature of a Nipype workflow are so called **iterables**. Iterables allow you to run a given workflow or subgraph several times with changing input values. For example, if you want to run an analysis pipeline on multiple subjects or with an FWHM smoothing kernel of 4mm, 6mm, and 8mm. This can easily be achieved with iterables and additionally allows you to do this all in parallel (simultaneous execution), if requested.

Go to the documentation section of Nipype's main homepage (http://nipype.readthedocs.io/en/latest/) to read more about MapNode, iterfield, and iterables (http://nipype.readthedocs.io/en/latest/users/mapnode_and_iterables.html), JoinNode, synchronize and itersource (http://nipype.readthedocs.io/en/latest/users/joinnode_and_itersource.html) and much more (http://nipype.readthedocs.io/en/latest/users/pipeline_tutorial.html). Nonetheless, a more detailed explanation will be given in a later section (http://miykael.github.io/nipype-beginner-s-guide/firstSteps.html#specify-workflows-connect-nodes) of this beginner's guide.

Note: For more practical and extended examples of Nipype concepts see Michael Waskom (https://github.com/mwaskom)'s really cool Jupyter notebooks about Interfaces (http://nbviewer.jupyter.org/github/mwaskom/nipype_concepts/blob/master/interfaces.ipynb), Iteration (http://nbviewer.jupyter.org/github/mwaskom/nipype_concepts/blob/master/iteration.ipynb) and Workflows (http://nbviewer.jupyter.org/github/mwaskom/nipype_concepts/blob/master/workflows.ipynb).

1.3.3 Execution Plugins

Plugins are components that describe how a workflow should be executed. They allow seamless execution across many architectures and make using parallel computation quite easy.

On a local machine, you can use the plugin **Serial** for a linear, or serial, execution of your workflow. If you machine has more than one core, you can use the **Multicore** plugin for parallel execution of your workflow. On a cluster, you have the option of using plugins for:

- **HTCondor**
- **PBS, Torque, SGE, LSF** (native and via IPython)
- **SSH** (via IPython)
- **Soma Workflow**

Note: Cluster operation often needs a special setup. You may wish to consult your cluster operators about which plugins are available.

To show how easily this can be done, the following code shows how to run a workflow with different plugins:

```python
# Normally calling run executes the workflow in series
workflow.run()

# But you can scale to parallel very easily.
# For example, to use multiple cores on your local machine
workflow.run('MultiProc', plugin_args={'n_procs': 4})

# or to other job managers
workflow.run('PBS', plugin_args={'qsub_args': '-q many'})
workflow.run('SGE', plugin_args={'qsub_args': '-q many'})
workflow.run('LSF', plugin_args={'qsub_args': '-q many'})
workflow.run('Condor')
workflow.run('IPython')

# or submit graphs as a whole
workflow.run('PBSGraph', plugin_args={'qsub_args': '-q many'})
workflow.run('SGEGraph', plugin_args={'qsub_args': '-q many'})
workflow.run('CondorDAGMan')
```

More about Plugins can be found on Nipype's main homepage under Using Nipype Plugins (http://nipype.readthedocs.io/en/latest/users/plugins.html).

CHAPTER

TWO

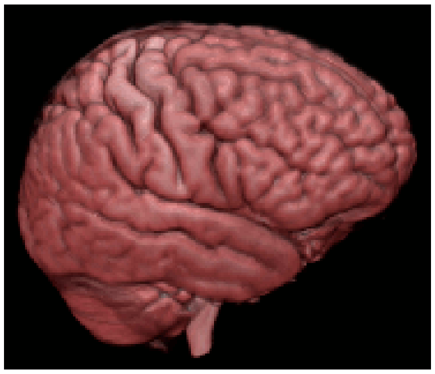
INTRODUCTION TO NEUROIMAGING

In this section, I will introduce you to the basics of analyzing neuroimaging data. I will give a brief explanation of how the neuroimaging data is acquired, how the data is prepared for analysis (also called preprocessing). Finally, I'll show how you can analyze your data using a model based on your hypothesis.

Note: This part is only a brief introduction to neuroimaging. Further information on this topic can be found under the Glossary (http://miykael.github.io/nipype-beginner-s-guide/glossary.html) and FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html) pages of this beginner's guide.

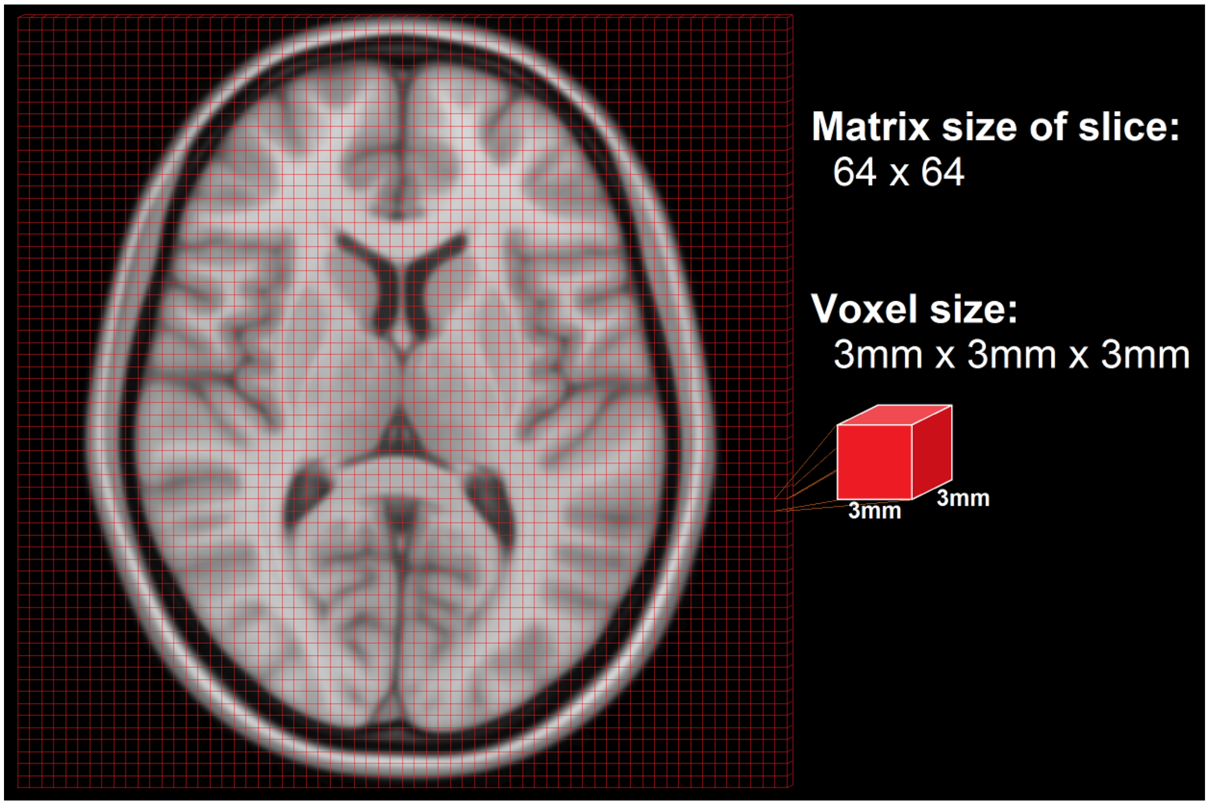
2.1 Acquisition of MRI Data

The technology and physics behind an MRI scanner is quite astonishing. But I won't go into the details of how it all works. You do need to know some terms, concepts, and parameters that are used to acquire MRI data in a scanning session and use that to construct useable images.



The brain occupies space, so when we collect data on how it fills space, we call that volume data, and all the volume data needed to create the complete, 3D image of the brain, recorded at one single timepoint and as pictured on the left, is called a **volume**. The data is measured in **voxels**, which are like the pixels used to display images on your screen, only in 3D. Each voxel has a specific dimension, in this case it is 1mm x 1mm x 1mm: a cube, so it is the same dimension from all sides (isotropic). Each voxel contains one value which stands for the average signal measured at the given location.

A standard anatomical volume, with an isotropic voxel resolution of 1mm contains almost 17 million voxels, which are arranged in a **3D matrix** of 256 x 256 x 256 voxels. The following picture shows a slice – one layer of the big, 3D matrix – through a brain volume, and the superimposed grid shows the remaining two dimensions of the voxels.



As the scanner can't measure the whole volume at once it has to measure portions of the brain sequentially in time. This is done by measuring one plane of the brain (generally the horizontal one) after the other. Such a plane is also called a **slice**. The **resolution** of the measured volume data, therefore, depends on the in-plane resolution (the size of the squares in the above image), the number of slices and their thickness (how many layers), and any possible gaps between the layers.

The quality of the measured data depends on the resolution and the following parameters:

- **repetition time (TR)**: time required to scan one volume
- **acquisition time (TA)**: time required to scan one slice. TA = TR - (TR/number of slices)
- **field of view (FOV)**: defines the extent of a slice, e.g. 256mm x 256mm

2.2 Specifics of MRI Data

MRI scanners output their neuroimaging data in a raw data format with which most analysis packages cannot work. **DICOM** is a common, standardized, raw medical image format, but the format of your raw data may be something else; e.g., **PAR/REC** format from Philips scanners. Raw data is saved in k-space (https://en.wikipedia.org/wiki/K-space_%28magnetic_resonance_imaging%29) format, and it needs to be converted into a format that the analysis packages can use. The most frequent format for newly generated data is called NIfTI (http://nifti.nimh.nih.gov/). If you are working with older datasets, you may encounter data in **Analyze** format. MRI data formats will have an **image** and a **header** part. For NifTI format, they are in the same file (.nii-file), whereas in the older Analyze format, they are in separate files (.img and .hdr-file).

- The **image** is the actual data and is represented by a 3D matrix that contains a value (e.g. gray value) for each voxel.
- The **header** contains information about the data like voxel dimension, voxel extend in each dimension, number of measured time points, a transformation matrix that places the 3D matrix from the **image** part in a 3D

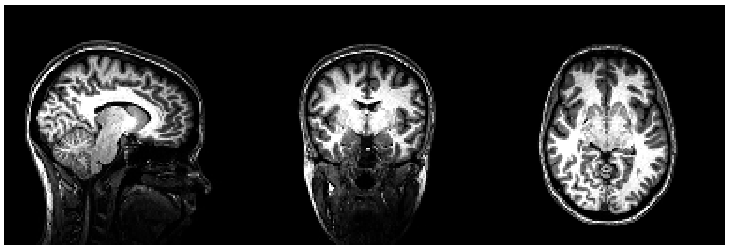
coordinate system, etc.

2.3 Modalities of MRI Data

There are many different kinds of acquisition techniques. But the most common ones are structural magnetic resonance imaging (**sMRI**), functional magnetic resonance imaging (**fMRI**) and diffusion tensor imaging (**DTI**).

2.3.1 sMRI (structural MRI)

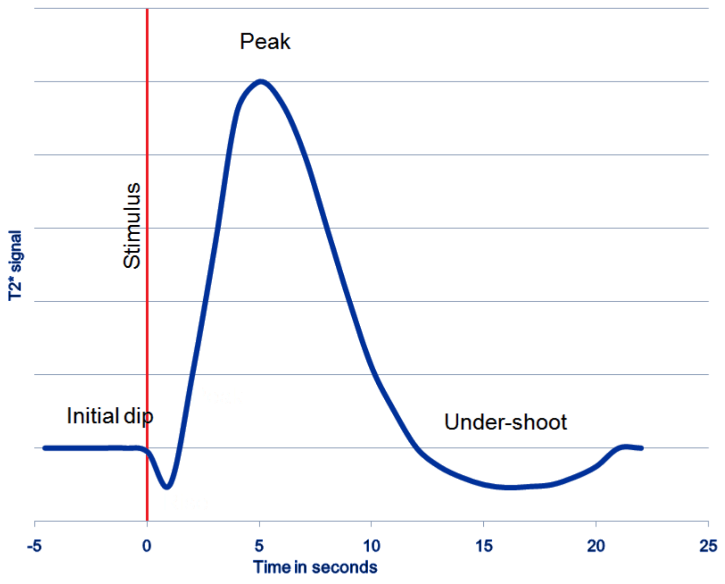
Structural magnetic resonance imaging (**sMRI**) is a technique for measuring the anatomy of the brain. By measuring the amount of water at a given location, sMRI is capable of acquiring a detailed anatomical picture of our brain. This allows us to accurately distinguish between different types of tissue, such as gray and white matter. Structural images are high-resolution images of the brain that are used as reference images for multiple purposes, such as corregistration, normalization, segmentation, and surface reconstruction.



As there is no time pressure during acquisition of anatomical images (the anatomy is not supposed to change while the person is in the scanner), a higher resolution can be used for recording anatomical images, with a voxel extent of 0.2 to 1.5mm, depending on the strength of the magnetic field in the scanner, e.g. 1.5T, 3T or 7T. Grey matter structures are seen in dark, and the white matter structures in bright colors.

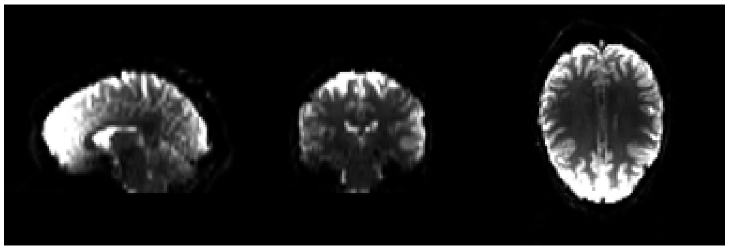
2.3.2 fMRI (functional MRI)

Functional magnetic resonance imaging (**fMRI**) is a technique for measuring brain activity. It works by detecting the changes in blood oxygenation and blood flow that occur in response to neural activity. Our brain is capable of so many astonishing things. But as nothing comes from nothing, it needs a lot of energy to sustain its functionality, and increased activity at a location increases the local energy consumption in the form of oxygen (O2) which is carried by the blood. Therefore, increased function results in increased blood flow towards the energy consuming location.



Immediately after neural activity the blood oxygen level decreases, known as the *initial dip*, because of the local energy consumption. This is followed by increased flow of new and oxygen-rich blood towards the energy consuming region. After 4-6 seconds a peak of blood oxygen level is reached. After no further neuronal activation takes place the signal decreases again and typically undershoots, before rising again to the baseline level.

The blood oxygen level is exactly what we measure with fMRI. The MRI Scanner is able to measure the changes in the magnetic field caused by the difference in the magnetic susceptibility of oxygenated (diamagnetic) and deoxygenated (paramagnetic) blood. The signal is therefore called the **Blood Oxygen Level Dependent (BOLD) response**.



Because the BOLD signal has to be measured quickly, the resolution of functional images is normally lower (2-4mm) than the resolution of structural images (0.5-1.5mm). But this depends strongly on the strength of the magnetic field in the scanner, e.g. 1.5T, 3T or 7T. In a functional image, the gray matter is seen as bright and the white matter as dark colors, which is the exact opposite to structural images.

Depending on the paradigm, we talk about **event-related**, **block** or **resting-state** designs.

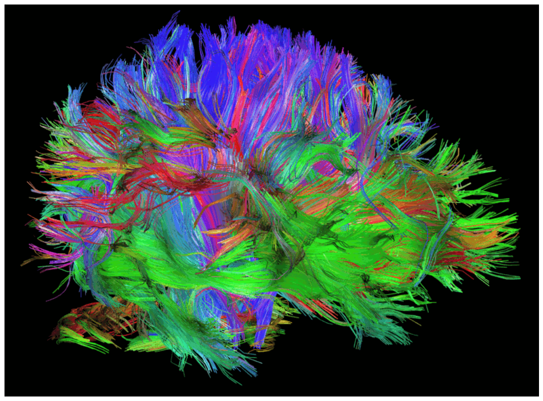
- **event-related design**: Event-related means that stimuli are administered to the subjects in the scanner for a short period. The stimuli are only administered briefly and generally in random order. Stimuli are typically visual, but audible or or other sensible stimuli could also be used. This means that the BOLD response consists of short bursts of activity, which should manifest as peaks, and should look more or less like the line shown in the graph above.

- **block design**: If multiple stimuli of a similar nature are shown in a block, or phase, of 10-30 seconds, that is a block design. Such a design has the advantages that the peak in the BOLD signal is not just attained for a short period but elevated for a longer time, creating a plateau in the graph. This makes it easier to detect an underlying activation increase.

- **resting-state design**: Resting-state designs acquire data in the absence of stimulation. Subjects are asked to lay still and rest in the scanner without falling asleep. The goal of such a scan is to record brain activation in the absence of an external task. This is sometimes done to analyze the functional connectivity of the brain.

2.3.3 dMRI (diffusion MRI)

Diffusion imaging is done to obtain information about the brain's white matter connections. There are multiple modalities to record diffusion images, such as diffusion tensor imaging (DTI), diffusion spectrum imaging (DSI), diffusion weighted imaging (DWI) and diffusion functional MRI (DfMRI). By recording the diffusion trajectory of the molecules (usually water) in a given voxel, one can make inferences about the underlying structure in the voxel. For example, if one voxel contains mostly horizontal fiber tracts, the water molecules in this region will mostly diffuse (move) in a horizontal manner, as they can't move vertically because of this neural barrier. The diffusion itself is mostly a Brownian motion (https://en.wikipedia.org/wiki/Brownian_motion).



There are many different diffusion measurements (https://en.wikipedia.org/wiki/Diffusion_MRI#Measures_of_anisotropy_and_diffusivi such as **mean diffusivity** (MD), fractional anisotropy (https://en.wikipedia.org/wiki/Fractional_anisotropy) (FA) and Tractography (https://en.wikipedia.org/wiki/Tractography). Each measurement gives different insights into the brain's neural fiber tracts. An example of a reconstructed tractography can be seen in the image to the left.

Diffusion MRI is a rather new field in MRI and still has some problems with its sensitivity to correctly detect fiber tracts and their underlying orientation. For example, the standard DTI method has almost no chance of reliably detecting kissing (touching) or crossing fiber tracts. To account for this disadvantage, newer methods such as **High-angular-resolution diffusion imaging** (HARDI) and Q-ball vector analysis were developed. For more about diffusion MRI see the Diffusion MRI Wikipedia page (https://en.wikipedia.org/wiki/Diffusion_MRI).

2.4 Analysis Steps

There are many different steps involved in a neuroimaging analysis and there is not just one order in which to perform them. Depending on the researcher, the paradigm at hand, or the modality analyzed (sMRI, fMRI, dMRI), the order can differ. Some steps may occur earlier or later or may be left out entirely. Nonetheless, the general procedure for fMRI analysis can be divided into the following three steps:

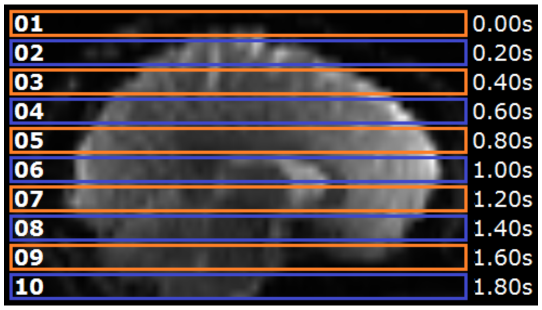
1. **Preprocessing**: Spatial and temporal preprocessing of the data to prepare it for the 1st and 2nd level inferential analysis
2. **Model Specification and Estimation**: Specifying and estimating parameters of the statistical model
3. **Statistical Inference**: Making inferences about the estimated parameters using appropriate statistical methods

2.4.1 Step 1: Preprocessing

Preprocessing is the term used to for all the steps taken to improve our data and prepare it for statistical analysis. We may correct or adjust our data for a number of things inherent in the experimental situation: to take account of time differences between acquiring each image slice, to correct for head movement during scanning, to detect 'artifacts' – anomalous measurements – that should be excluded from subsequent analysis; to align the functional images with the reference structural image, and to normalize the data into a standard space so that data can be compared among several subjects; to apply filtering to the image to increase the signal-to-noise ratio; finally, if sMRI is intended, a segmentation step may be performed. We will now look at each of those steps in more detail.

Slice Timing Correction (fMRI only)

Because functional MRI measurement sequences don't acquire every slice in a volume at the same time we have to account for the time differences among the slices. For example, if you acquire a volume with 37 slices in ascending order, and each slice is acquired every 50ms, there is a difference of 1.8s between the first and the last slice acquired. You must know the order in which the slices were acquired to be able to apply the proper correction. Slices are typically acquired in one of three methods: descending order (top-down); ascending order (bottom-up); or interleaved (acquire every other slice in each direction), where the interleaving may start at the top or the bottom. (Left: *ascending*, Right: *interleaved*)



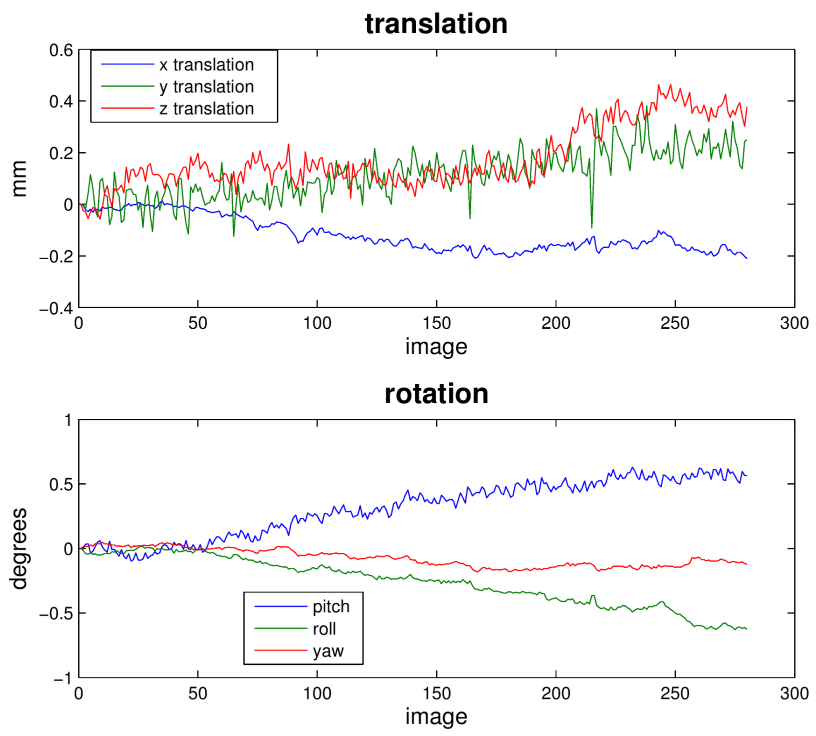
Slice Timing Correction is used to compensate for the time differences between the slice acquisitions by temporally interpolating the slices so that the resulting volume is close to equivalent to acquiring the whole brain image at a single time point. This temporal factor of acquisition especially has to be accounted for in fMRI models where timing is an important factor (e.g. for event related designs, where the type of stimulus changes from volume to volume).

Motion Correction (fMRI only)

Motion correction, also known as Realignment, is used to correct for head movement during the acquisition of functional data. Even small head movements lead to unwanted variation in voxels and reduce the quality of your data. Motion correction tries to minimize the influence of movement on your data by aligning your data to a reference time volume. This reference time volume is usually the mean image of all timepoints, but it could also be the first, or some other, time point.

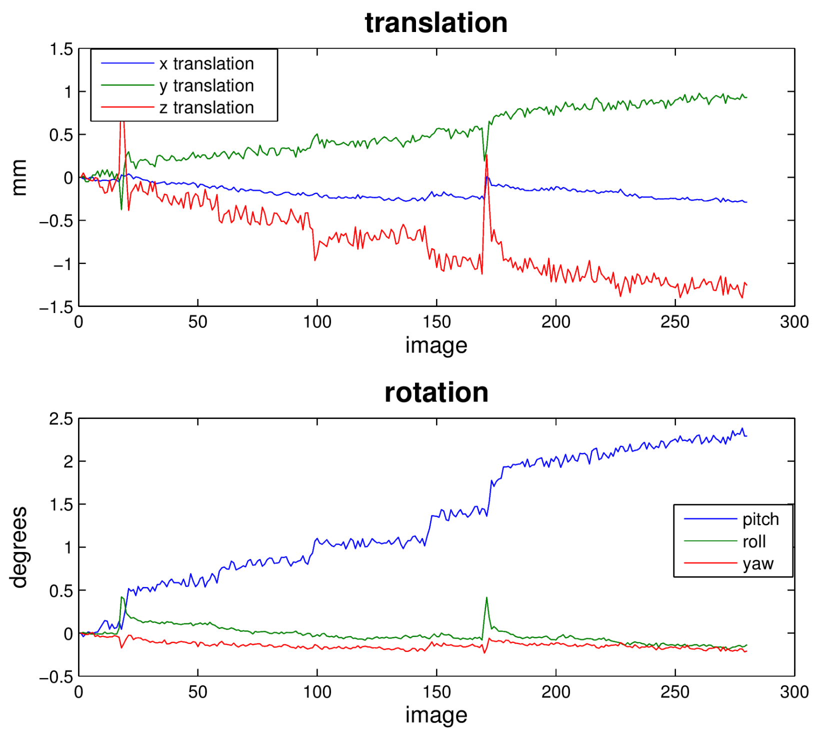
Head movement can be characterized by six parameters: Three translation parameters which code movement in the directions of the three dimensional axes, movement along the X, Y, or Z axes; and three rotation parameters which code rotation about those axes, rotation centered on each of the X, Y, and Z axes).

Realignment usually uses an affine rigid body transformation to manipulate the data in those six parameters. That is, each image can be moved but not distorted to best align with all the other images. Below you see a plot of a "good" subject where the movement is minimal.



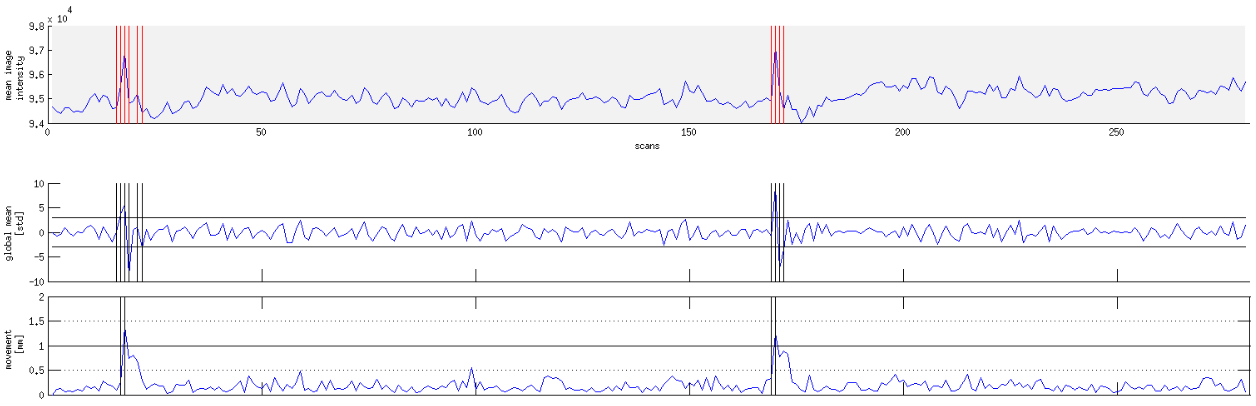
Artifact Detection (fMRI only)

Almost no subjects lie perfectly still. As we can see from the sharp spikes in the graphs below, some move quite drastically. Severe, sudden movement can contaminate your analysis quite severely.



Motion correction tries to correct for smaller movements, but sometimes it's best to just remove the images acquired during extreme rapid movement. We use **Artifact Detection** to identify the timepoints/images of the functional image that vary so much they should be excluded from further analysis and to label them so they are excluded from subsequent analyses.

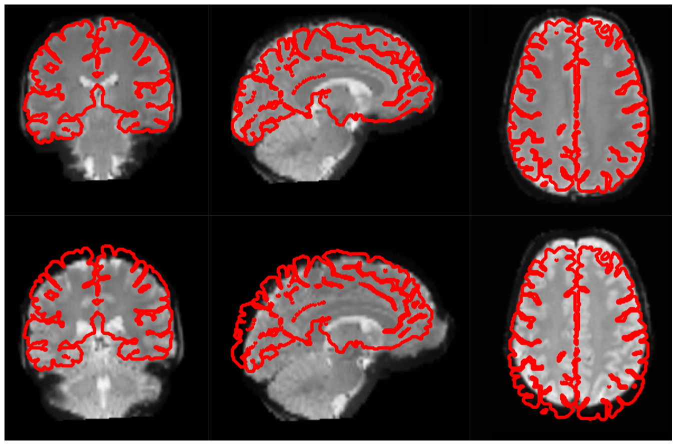
For example, checking the translation and rotation graphs for a session shown above for sudden movement greater than 2 standard deviations from the mean, or for movement greater than 1mm, artifact detection would show that images 16-19, 21, 22 and 169-172 should be excluded from further analysis. The graph produced by artifact detection, with vertical lines corresponding to images with drastic variation is shown below.



Coregistration

Motion correction aligns all the images within a volume so they are 'aligned'. Coregistration aligns the functional image with the reference structural image. If you think of the functional image as having been printed on tracing paper, coregistration moves that image around on the reference image until the alignment is at its best. In other words, coregistration tries to superimpose the functional image perfectly on the anatomical image. This allows further transformations of the anatomical image, such as normalization, to be directly applied to the functional image.

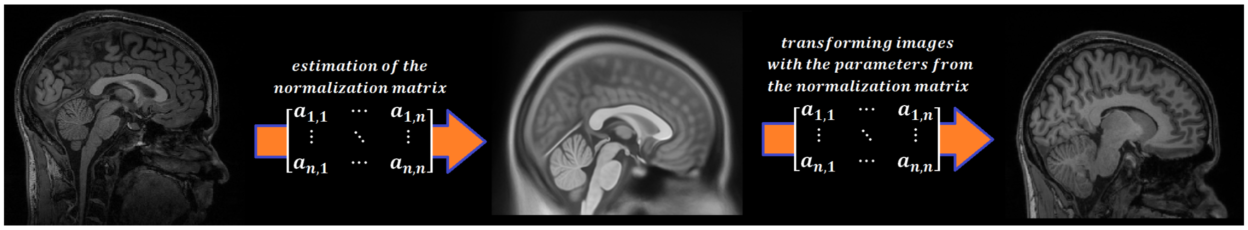
The following picture shows an example of good (top) and bad (bottom) coregistration of functional images with the corresponding anatomical images. The red lines are the outline of the cortical folds of the anatomical image superimposed on the underlying greyscale functional image.



Normalization

Every person's brain is slightly different from every other's. Brains differ in size and shape. To compare the images of one person's brain to another's, the images must first be translated onto a common shape and size, which is called **normalization**. Normalization maps data from the individual subject-space it was measured in onto a reference-space.

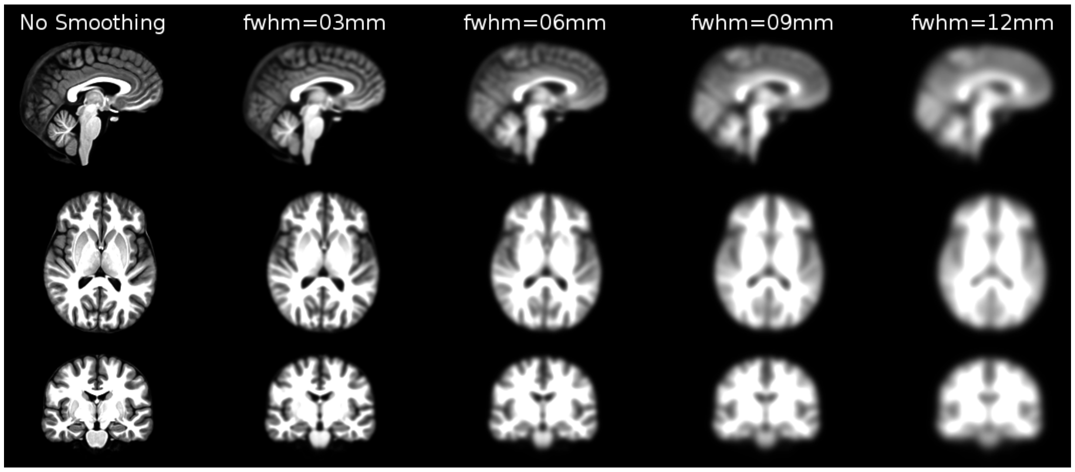
Once this step is completed, a group analysis or comparison among data can be performed. There are different ways to normalize data but it always includes a template and a source image.


estimation of the normalization matrix
$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$
transforming images with the parameters from the normalization matrix
$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$

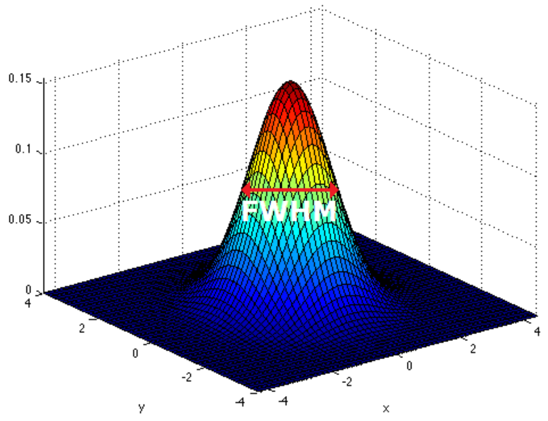
- The **template** image is the standard brain in reference-space onto which you want to map your data. This can be a Talairach-, MNI-, or SPM-template, or some other reference image you choose to use.
- The **source** image (normally a higher resolution structural image) is used to calculate the transformation matrix necessary to map the source image onto the template image. This transformation matrix is then used to map the rest of your images (functional and structural) into the reference-space.

Smoothing

Structural as well as functional images are smoothed by applying a filter to the image. Smoothing increases the signal to noise ratio of your data by filtering the highest frequencies from the frequency domain; that is, removing the smallest scale changes among voxels. That helps to make the larger scale changes more apparent. There is some inherent variability in functional location among individuals, and smoothing helps to reduce spatial differences between subjects and therefore aids comparing multiple subjects. The trade-off, of course, is that you lose resolution by smoothing. Keep in mind, though, that smoothing can cause regions that are functionally different to combine with each other. In such cases a surface based analysis with smoothing on the surface might be a better choice.



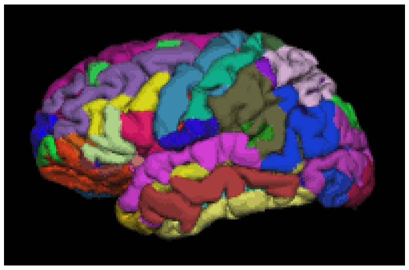
Smoothing is implemented by applying a 3D Gaussian kernel to the image, and the amount of smoothing is typically determined by its full width at half maximum (**FWHM**) parameter. As the name implies, FWHM is the width/diameter of the smoothing kernel at half of its height. Each voxel's value is changed to the result of applying this smoothing kernel to its original value.



Choosing the size of the smoothing kernel also depends on your reason for smoothing. If you want to study a small region, a large kernel might smooth your data too much. The filter shouldn't generally be larger than the activation you're trying to detect. Thus, the amount of smoothing that you should use is determined partly by the question you want to answer. Some authors suggest using twice the voxel dimensions as a reasonable starting point.

Segmentation (sMRI only)

Segmentation is the process by which a brain is divided into neurological sections according to a given template specification. This can be rather general, for example, segmenting the brain into gray matter, white matter and cerebrospinal fluid, as is done with SPM's Segmentation, or quite detailed, segmenting into specific functional regions and their subregions, as is done with FreeSurfer's `recon-all`, and that is illustrated in the figure.



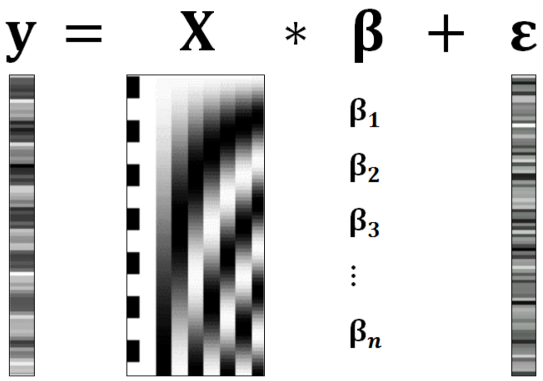
Segmentation can be used for different things. You can use the segmentation to aid the normalization process or use it to aid further analysis by using a specific segmentation as a mask or as the definition of a specific region of interest (ROI).

2.4.2 Step 2: Model Specification and Estimation

To test our hypothesis on our data we first need to specify a model that incorporates this hypothesis and accounts for multiple factors such as the expected function of the BOLD signal, the movement during measurement, experiment specify parameters and other regressors and covariates. Such a model is usually represented by a Generalized Linear Model (GLM).

The General Linear Model

A GLM describes a response (y), such as the BOLD response in a voxel, in terms of all its contributing factors (xβ) in a linear combination, whilst also accounting for the contribution of error (ϵ). The column (y) corresponds to one voxel and one row in this column corresponds to one time-point.



- **y = dependent variable** observed data (e.g. BOLD response in a single voxel)
- **X = Independent Variable (aka. Predictor)** e.g. *experimental conditions* (embodies all available knowledge about experimentally controlled factors and potential confounds), *stimulus information* (onset and duration of stimuli), *expected shape of BOLD response*
- β **= Parameters (aka regression coefficient/beta weights)** Quantifies how much each predictor (X) independently influences the dependent variable (Y)
- ϵ **= Error** Variance in the data (Y) which is not explained by the linear combination of predictors ($X\beta$). The error is assumed to be normally distributed.

The predictor variables are stored in a so called **Design Matrix**. The β parameters define the contribution of each component of this design matrix to the model. They are estimated so as to minimize the error, and are used to generate the **contrasts** between conditions. The **Errors** is the difference between the observed data and the model defined by Xβ.

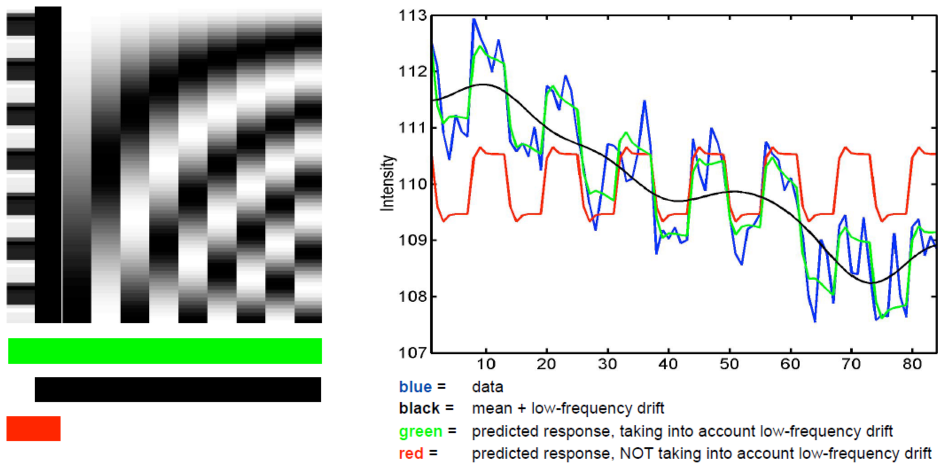
Potential problems of the GLM approach

BOLD responses have a delayed and dispersed form

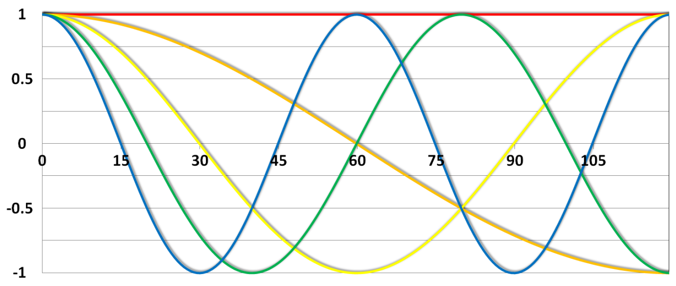
- We have to take the time delay and the HRF shape of the BOLD response into account when we create our design matrix.

BOLD signals include substantial amounts of low-frequency noise

- By high pass filtering our data and adding time regressors of 1st, 2nd,... order we can correct for low-frequency drifts in our measured data. This low frequency signals are caused by non-experimental effects, such as scanner drift etc.



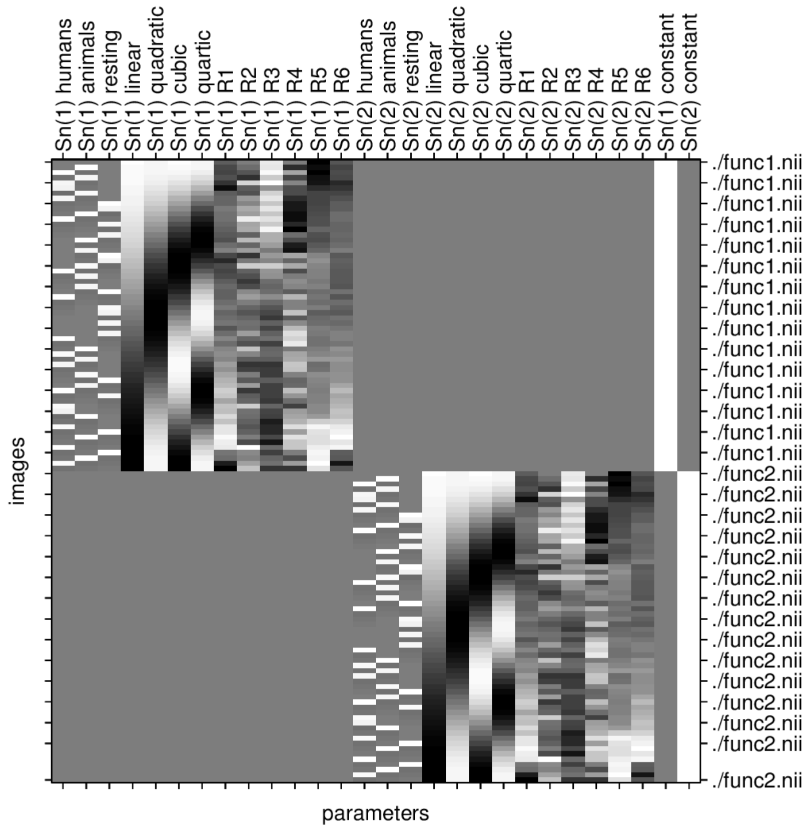
This **High pass Filter** is established by setting up discrete cosine functions over the time period of your acquisition. In the example below you see a constant term of 1, followed by half of a cosine function increasing by half a period for each following curve. Such regressors correct for the influence of changes in the low-frequency spectrum.



Example of a Design Matrix

Let us assume we have an experiment where we present subjects faces of humans and animals alike. Our goal is to measure the difference between the brain activation when a face of an animal is presented in contrast to the activation of the brain when a human face is presented. Our experiment is set up in such a way that subjects have two different blocks of stimuli presentation. In both blocks there are timepoints where faces of humans, faces of animals and no faces (resting state) are presented.

Now, we combine all that we know about our model into one single Design Matrix. This Matrix contains multiple columns, which contain information about the stimuli (onset, duration and curve function of the BOLD-signal i.e. the shape of the HRF). In our example column *Sn(1) humans* and *Sn(1) animals* code for the stimuli of humans and animals during the first session of our fictive experiment. Accordingly, Sn(2) codes for all the regressors in the second session. *Sn(1) resting* codes for the timepoints where subjects weren't presented any stimuli.



The y-axis codes for the measured scan or the passed time, depending on the specification of your design. The x-axis stands for all the regressors that we specified.

The regressors *Sn(1) R1* to *Sn(1) R6* stand for the movement parameters we got from the realignment process. The regressors *Sn(1) linear*, *Sn(1) quadratic*, *Sn(1) cubic* and *Sn(1) quartic* are just examples of correction for the low frequency in your data. If you are using a high-pass filter of e.g. 128 seconds you don't need to specifically include those regressors in your design matrix.

Note: Adding one more regressors to your model decrease the degrees of freedom in your statistical tests by one.

Model Estimation

After we specified the parameters of our model in a design matrix we are ready to estimate our model. This means that we apply our model on the time course of each and every voxel.

Depending on the software you are using you might get different types of results. If you are using **SPM** the following images are created each time an analysis is performed (1st or 2nd level):

- **beta images** images of estimated regression coefficients (parameter estimate). beta images contain information about the size of the effect of interest. A given voxel in each beta image will have a value related to the size of effect for that explanatory variable.
- **error image - `ResMS`-image** residual sum of squares or variance image. It is a measure of within-subject error at the 1st level or between-subject error at the 2nd level analysis. This image is used to produce spmT images.
- **con images - `con`-images** during contrast estimation beta images are linearly combined to produce relevant `con`-images

- **T images - `spmT`-images** during contrast estimation the beta values of a `con`-image are combined with error values of the `ResMS`-image to calculate the t-value at each voxel

2.4.3 Step 3: Statistical Inference

Before we go into the specifics of a statistical analysis, let me explain you the difference between a 1st and a 2nd level analysis.

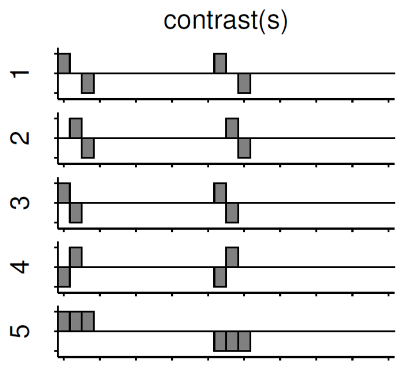
1st level analysis (within-subject) A 1st level analysis is the statistical analysis done on each and every subject by itself. For this procedure the data doesn't have to be normalized, i.e in a common reference space. A design matrix on this level controls for subject specific parameters as movement, respiration, heart beat, etc.

2nd level analysis (between-subject) A 2nd level analysis is the statistical analysis done on the group. To be able to do this, our subject specific data has to be normalized and transformed from subject-space into reference-space. Otherwise we wouldn't be able to compare subjects between each other. Additionally, all contrasts of the 1st level analysis have to be estimated because the model of the 2nd level analysis is conducted on them. The design matrix of the 2nd level analysis controls for subject specific parameters such as age, gender, socio-economic parameters, etc. At this point we also specify the group assignment of each subject.

Contrast Estimation

Independent of the level of your analysis, after you've specified and estimated your model you now have to estimate the contrasts you are interested in. In such a **contrast** you specify how to weight the different regressors of your design matrix and combine them in one single image.

For example, if you want to compare the brain activation during the presentation of human faces compared to the brain activation during the presentation of animal faces over two sessions you have to weight the regressors *Sn(1) humans* and *Sn(2) humans* with 1 and *Sn(1) animals* and *Sn(2) animals* with -1, as can be seen in **contrast 3**. This will subtract the value of the animal-activation from the activation during the presentation of human faces. The result is an image where the positive activation stands for "more active" during the presentation of human faces than during the presentation of animal faces.

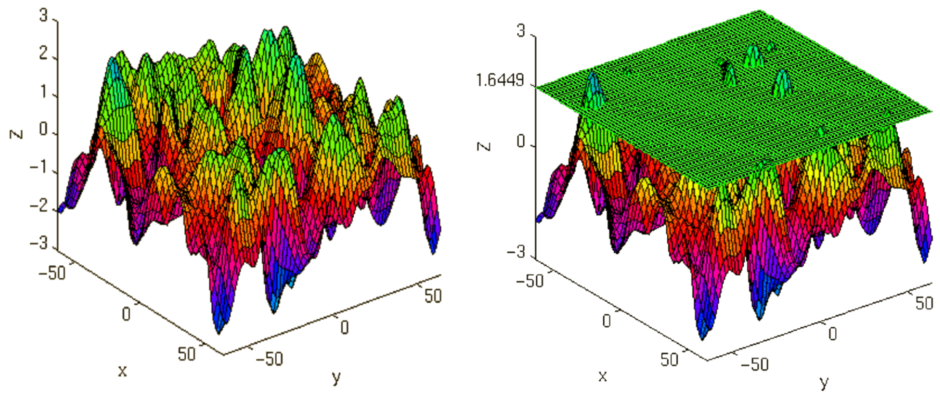


Contrast 1 codes for *human faces vs. resting*, contrast 2 codes for *animal faces vs. resting*, contrast 4 codes for *animal faces vs. human faces* (which is just the inverse image of contrast 3) and contrast 5 codes for *session 1 vs. session 2*, which looks for regions which were more active in the first session than in the second session.

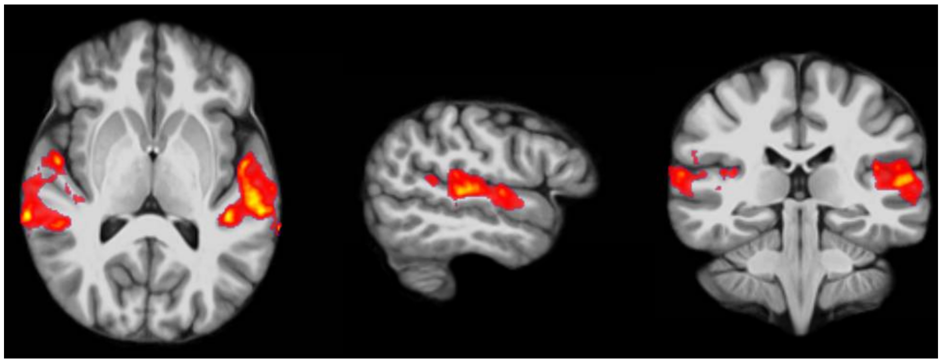
Thresholding

After the contrasts are estimated there is only one final step to be taken before you get a scientific based answer to your question. You have to threshold your results. With that I mean, you have to specify the level of significance you want to test your data on, you have to correct for multiple comparison and you have to specify the parameters of the results you are looking for. E.g.:

- **FWE-correction**: The family-wise error correction is one way to correct for multiple comparisons
- **p-value**: specify the hight of the significance threshold that you want to use (e.g. z=1.6449 equals p<0.05 (one-tailed); see image)
- **voxel extend**: specify the minimum size of a "significant" cluster by specifying the number of voxel it at least has to contain.



If you do all this correctly, you'll end up with something as shown in the following picture. The picture shows you the average brain activation of 20 subjects during the presentation of an acoustic stimuli. The p-value are shown from red to yellow, representing values from 0.05 to 0.00. Shown are only cluster with a voxel extend of at least 100 voxels.



CHAPTER

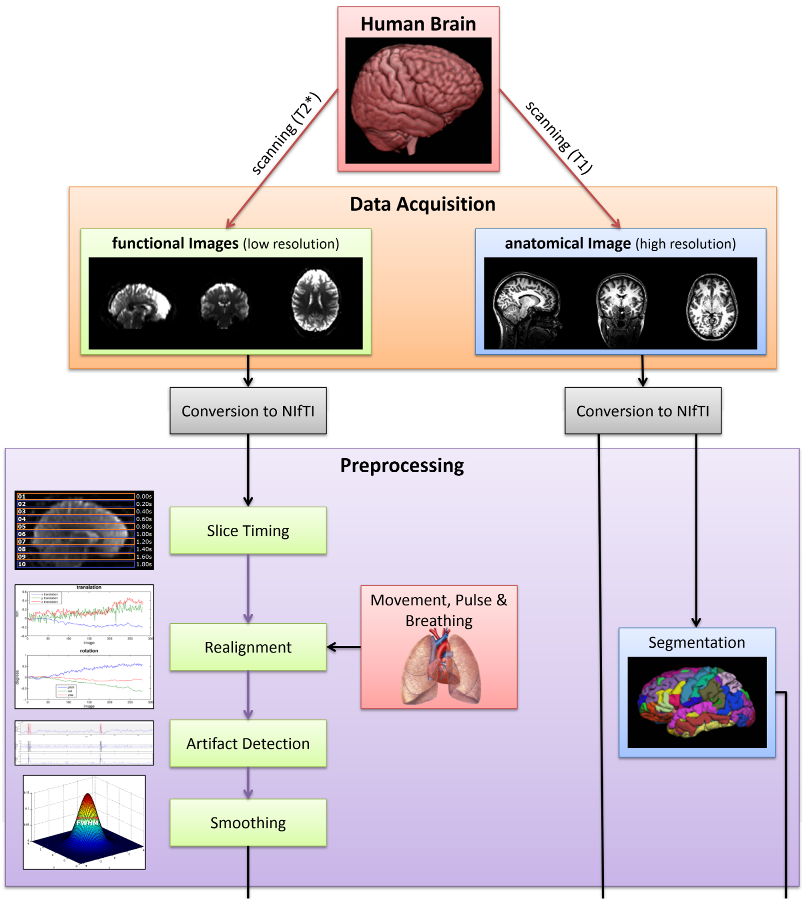
THREE

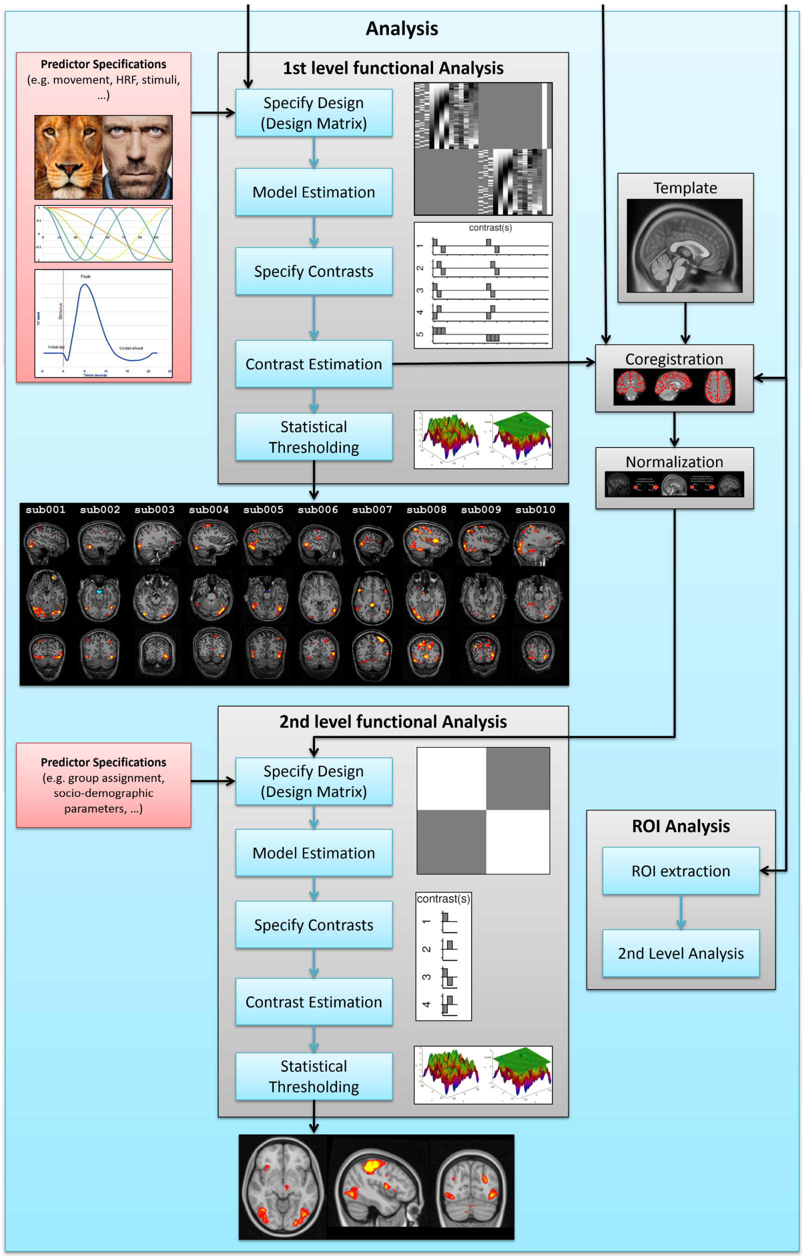
NIPYPE AND NEUROIMAGING

As you've seen in the previous chapter, there are many steps involved in the analysis of neuroimaging data. And there are even more possibilities to combine them. And adding to all this complexity, there are often numerous different software packages for each step. This is where Nipype can help you. Changing the order of preprocessing or analysis steps is as simple as changing the flow of a workflow.

3.1 Neuroimaging Workflow

Let's get back to the steps involved in the analysis of fMRI data from the previous chapter. Keep in mind that this is only one possible way of preprocessing and analyzing fMRI data. But if we connect up all the different steps into one big workflow we end up with the following structure.



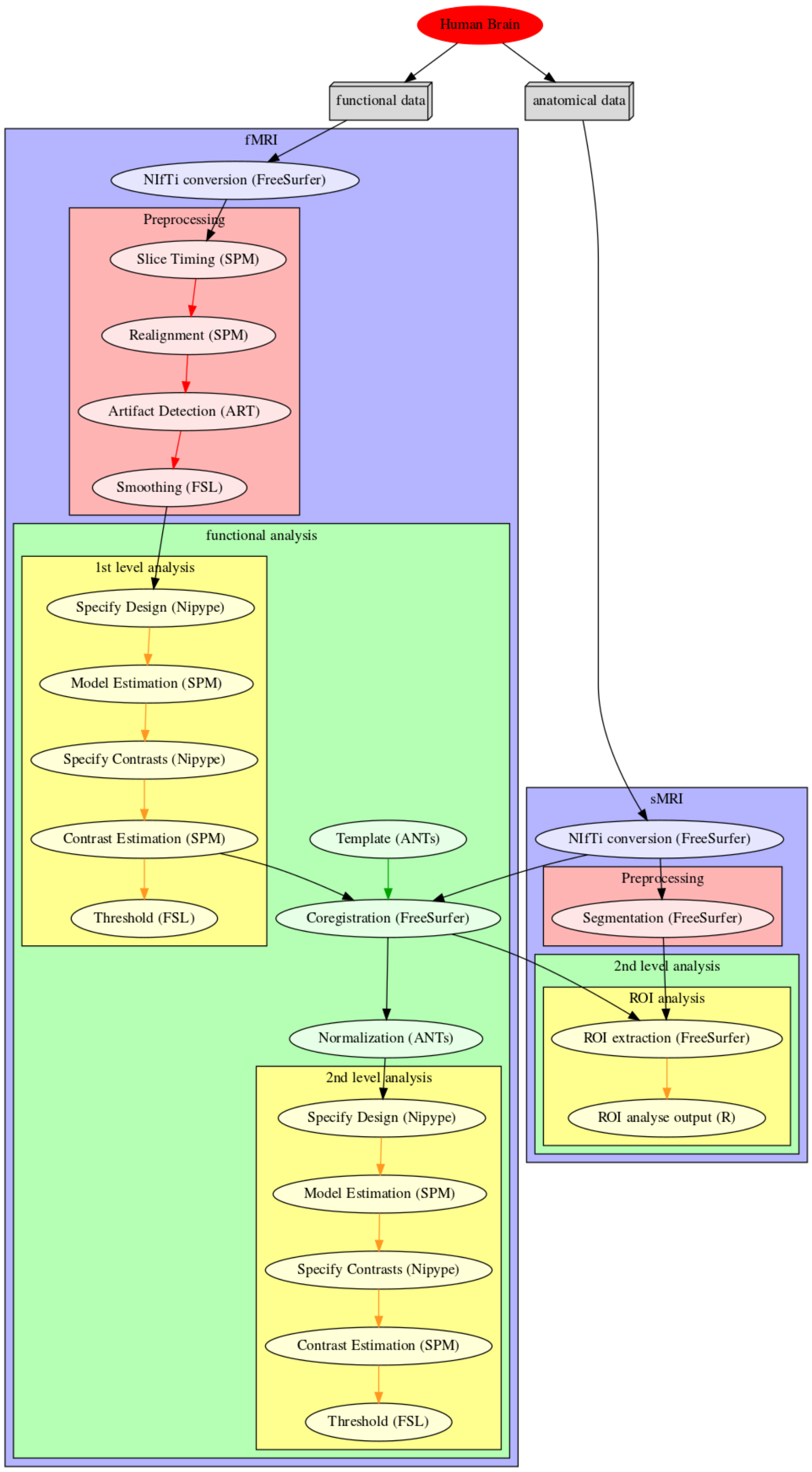

Analysis
Predictor Specifications (e.g. movement, HRF, stimuli, ...)
1st level functional Analysis
Specify Design (Design Matrix)
Model Estimation
Specify Contrasts
Contrast Estimation
Statistical Thresholding
contrast(s)
Template
Coregistration
Normalization
sub001 sub002 sub003 sub004 sub005 sub006 sub007 sub008 sub009 sub010
2nd level functional Analysis
Predictor Specifications (e.g. group assignment, socio-demographic parameters, ...)
Specify Design (Design Matrix)
Model Estimation
Specify Contrasts
Contrast Estimation
Statistical Thresholding
ROI Analysis
ROI extraction
2nd Level Analysis

This all seems to be really big and complex. And what if you want to first want to do a motion correction and then a slice timing correction? Or add an additional step into an already established analysis. With Nipype, this is very easy.

3.2 Nipype Workflow

Nipype enables you to create the exact workflow that you want and gives you the opportunity to switch between the software packages (e.g. *FreeSurfer*, *FSL*, *SPM*, *ANTs*, *AFNI*,...) as you like. The power to analyze your data exactly as you want it lies in your hand.

Let's get back to the workflow above. In the world of Nipype, this neuroimaging workflow would look something like this:



As you can see, each step of the process is represented by a node in the workflow (e.g. *Motion Correction*, *Coregistration*, *Normalization*, *Smoothing*, *Model Estimation*). And each of those nodes can come from a different software package (e.g. *FreeSurfer*, *FSL*, *SPM*, *ANTs*, *AFNI*, *Nipype*). The freedom to chose the software package to use, to guide the flow and sequential order of the execution is completely up to you. Even if you want to run a node with different parameters (e.g. *fwhm = 4 and 8*) this can be done with no problem. And the great thing about all this. All of those steps can be done in parallel!

If you understand those concepts, you will be able to use Nipype in no time. Because this is all there is to know about Nipype. But before we'll start with the first Nipype script, lets first make sure that your system is set up correctly. All about how to check that and how to install all required softwares can be found in the next chapter of this Beginner's Guide.

Note: 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 you can look it up at various places on the Nipype homepage (http://nipype.readthedocs.io/en/latest/). A lot of very good tutorials and examples about the usage of workflows for specific situations can be found here: Tutorials and Examples (http://nipype.readthedocs.io/en/latest/users/pipeline_tutorial.html).

CHAPTER

FOUR

DOWNLOAD AND INSTALL NIPYPE

All you need to know to download and install Nipype can be found on the official homepage under Download and Install (http://nipype.readthedocs.io/en/latest/users/install.html). There you find a link to the newest version and more information about which dependencies are necessary or recommended.

Installing Nipype as described on the official homepage should be rather easy for most users. There are some tricky steps that sometimes seem to be less straight forward. But don't worry, I've written my own installation guide to help and assist users with less UNIX experience. The following instructions hopefully help everybody to install and run Nipype on their system. Note that some steps strongly depend on the system you are using and it might be possible that some dependencies are already installed on your system. Also,

Note: It is highly recommended to run Nipype on a machine with either a Mac or Linux OS, as most dependencies such as FSL or FreeSurfer don't run on Windows. If you have a Windows machine, I recommend you to install Ubuntu (http://www.ubuntu.com/download/desktop) on your system.

The following steps describe how I was able to set up and run Nipype on a System with a newly installed Ubuntu 14.04 LTS OS (64-bit), called "*trusty*". To people with older Ubuntu version, don't worry, Nipype does also run on much older versions, as long as they are not too ancient. To check which version of Ubuntu you are running use the command `lsb_release -d`. You don't have to upgrade to version 14.04 if you have an older version as long as it isn't too ancient.

To Mac Users: I haven't installed Nipype on a Mac, yet. But the steps should be almost identical. If any major differences occur, please let me know.

4.1 Prepare your System

Before we start, make sure that your Ubuntu system is up to date with the following command:

```
#Update and upgrade your system
sudo apt-get update && sudo apt-get upgrade
```

4.2 Anaconda

As you can see under Dependencies (http://nipype.readthedocs.io/en/latest/users/install.html#recommended-software), there are a lot of software packages that have to be installed on your system to run Nipype. For example: *python*, *ipython*, *matplotlib*, *networkx*, *numpy*, *scipy*, *sphinx* etc. Luckily, most of the those required dependencies, most importantly a working Python environment, can be set up by installing Anaconda (https://www.continuum.io/downloads).

4.2.1 Install Anaconda

Instructions on how to install Anaconda can be found on the official homepage (http://docs.continuum.io/anaconda/install). Following are the steps how I've installed the Anaconda package on my system:

1. Download the software under https://www.continuum.io/downloads and move the downloaded sh-file to your Download folder. In my case this folder is at `/home/username/Downloads` abbreviated with `~/Downloads`.
2. Install the downloaded sh-file with the following command:

```
bash ~/Downloads/Anaconda-2.1.0-Linux-x86_64.sh
```

3. Specify the location of the installation (default is ok).
4. Add the anaconda binary directory to your PATH environment variable. This path is automatically added if you answer the question *"Do you wish the installer to prepend the Anaconda install location to PATH in your /home/username/.bashrc ? [yes|no]"* with yes. Otherwise, add the following line to your *.bashrc* file:

```
export PATH=/home/username/anaconda/bin:$PATH
```

Note: `.bashrc` is read and executed whenever you run bash using an interactive shell, i.e. the terminal. This file usually is stored in your home folder at `/home/username/.bashrc` or in other words `~/.bashrc`. The `export` command tells your system to export a variables from the specified package so that they can be used inside the current shell. For example, `export PATH=/home/somepath:$PATH` inserts `/home/somepath` to the beginning of the variable PATH and exports it.

4.2.2 Update Anaconda

To update anaconda to the newest version and to clean unused and older content use the following command:

```
conda update conda && conda update anaconda && conda clean --packages --tarballs
```

Now make sure that you have all Nipype required dependencies up to date with the following command:

```
conda update python ipython ipython-notebook matplotlib \
             networkx numpy scipy sphinx traits dateutil nose pydot
```

Note: To update a software package in anaconda use the command "conda update packagename". For example, if you want to update python use "conda update python"

4.2.3 Test Anaconda

Now that Anaconda is installed let's test if our python environment is ready to run.

1. Open a new terminal and type in the command `ipython`. This should bring you to the IPython environment. IPython is used to run all your python scripts. Fore more information about Python and IPython see the support section (http://miykael.github.io/nipype-beginner-s-guide/links.html) of this beginner's guide.
2. To check if everything is set up correctly try to import numpy with the following command:

```
import numpy
```

If you see no *ImportError* message, everything is fine and we can get on to the next step.

4.3 NeuroDebian

To facilitate the installation of some necessary and recommended software packages such as FSL and Nipype itself, Debian and Ubuntu based system should install the NeuroDebian (http://neuro.debian.net/) repository. To see which software packages are included in NeuroDebian, go to NITRCT - NeuroDebian (http://www.nitrc.org/projects/neurodebian/).

1. To install NeuroDebian on your System go to the Get NeuroDebian (http://neuro.debian.net/#get-neurodebian) and select the operating system and the server you want to use. In my case, the operating system is *'Ubuntu 14.04 "Trusty Tahr" (trusty)'*. If you have an Ubuntu OS but don't know which version, just type *lsb_release -a* in the terminal and it will show you.
2. Chose the option "All software"
3. Now you should see two lines of command. In my case they were the following:

   ```
   wget -O- http://neuro.debian.net/lists/trusty.de-md.full | sudo tee /etc/
   ↪apt/sources.list.d/neurodebian.sources.list
   sudo apt-key adv --recv-keys --keyserver pgp.mit.edu 2649A5A9
   ```

   Run those two lines of code in your terminal.
4. After all this is done, update your system with the following command: `sudo apt-get update`

Now you are read to install Nipype, FSL, AFNI and more.

Note: If you have problem with the `wget` command in the 3rd step it is most likely because of the root permission (the sudo command in the second half of the command). When the wget command seems to halt and do nothing type in your password and it should go on.

4.4 Nipype

4.4.1 Install Nipype

Finally, it's time to install Nipype. There are multiple ways how you can install Nipype, but assuming that you installed anaconda beforehand, the easiest way to install Nipype is by using conda. To do this you need to add the channel `conda-forge` to your channels:

```
conda config --add channels conda-forge
```

Once the conda-forge channel has been enabled, nipype can be installed with:

```
conda install nipype
```

As an alternative, you could also install Nipype with either `pip install -e git+https://github.com/nipy/nipype#egg=nipype` or `easy_install nipype`. For more information about the installation from sourcecode, go to the main page (http://nipype.readthedocs.io/en/latest/users/install.html).

4.4.2 Install Pyhon Dependencies

There are a few additional python dependencies that you cannot install via anaconda, such as: nibabel, rdflib, nipy, dipy and graphviz. To install those, use the following command:

```
#Install packages with pip
pip install nibabel rdflib nipy dipy

#Install graphviz and pygraphviz separately
sudo apt-get install graphviz libgraphviz-dev
pip install --upgrade pygraphviz graphviz
```

4.4.3 Test Nipype

To test if everything worked fine and if you're able to use Nipype go into an IPython environment and import nipype with the command: `import nipype`. If you see no *ImportError* message, everything is set up correctly.

4.4.4 Upgrade Nipype (and other python dependencies)

If you want to be sure that you have the newest version or update a certain package use the `pip install` command with the flag `--upgrade`. So, if you want to upgrade Nipype to the newest version use the following command:

```
pip install --upgrade nipype
```

If you want to upgrade all other required python dependencies as well use the following command:

```
pip install --upgrade nibabel nipype rdflib nipy dipy pygraphviz graphviz
```

4.4.5 Upgrade Nipype to the developer version

If you want or have to upgrade Nipype to the developer version us the following steps. Such an upgrade is only recommended to people who know what they are doing or need a certain fix that isn't distributed yet in the general Nipype version.

The most current developer version of Nipype can be found on GitHub (https://github.com/) under Nipype @ GitHub (https://github.com/nipy/nipype). The following steps assume assume that you've already set up your own GitHub account and are ready to download the Nipype repository:

1. First, open a terminal and download the Nipype repository at the current location with `git clone https://github.com/nipy/nipype.git`, or download the repository directly by using this link (https://codeload.github.com/nipy/nipype/zip/master).
2. The just downloaded nipype folder contains another folder called `nipype`. This is the folder that contains the newest version of Nipype.
3. Now, either add the path to this folder to the `PYTHONPATH` list (make sure that `PYTHONPATH` only contains one Nipype folder) or delete the current nipype folder and move the new github `nipype` folder to this location. This can be done with the following command:

   ```
   rm -rf ~/anaconda/lib/python2.7/site-packages/nipype
   cp -R ~/Downloads/nipype/nipype ~/anaconda/lib/python2.7/site-packages/
   ↪nipype
   ```

Note: If you haven't set up a GitHub account yet but don't know how to set everything up, see this link: Set Up Git (https://help.github.com/articles/set-up-git).

CHAPTER

FIVE

DOWNLOAD AND INSTALL INTERFACES

5.1 FSL

5.1.1 Download and Installation

FSL (http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/) is a comprehensive library of analysis tools for fMRI, MRI and DTI data. An overview of FSL's tools can be found on their homepage under FSL Overview (http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/FslOverview).

The installation of FSL is simple if you've already installed the NeuroDebian repository.

Just run the following command:

```
sudo apt-get install fsl
```

Otherwise, go through the official FSL installation guide (http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/FslInstallation).

5.1.2 Configuration

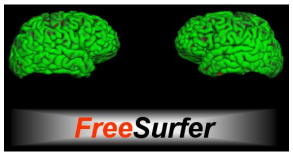
Before you can run FSL, your system first needs to know where the software is installed at. On a ubuntu system, this is usually under `/usr/share/fsl`. Therefore, add the following code to your *.bashrc* file. (To open and edit your *.bashrc* file on Ubuntu, us the following command: `gedit ~/.bashrc`)

```
#FSL
FSLDIR=/usr/share/fsl
. ${FSLDIR}/5.0/etc/fslconf/fsl.sh
PATH=${FSLDIR}/5.0/bin:${PATH}
export FSLDIR PATH
```

5.1.3 Test FSL

To test if FSL is correctly installed, open a new terminal and type in the command `fsl`. If everything was set up correctly you should see the FSL GUI with the version number in the header. In my case this is version 5.0.7.

5.2 FreeSurfer



FreeSurfer (http://surfer.nmr.mgh.harvard.edu/) is an open source software suite for processing and analyzing (human) brain MRI images. The installation of FreeSurfer (http://surfer.nmr.mgh.harvard.edu/) includes abit more steps than the other installations, but the official FreeSurfer: Download and Install (http://surfer.nmr.mgh.harvard.edu/fswiki/DownloadAndInstall) homepage is written very well and should get you through it without any problem. Nonetheless, following are the steps how I've installed FreeSurfer on my system.

5.2.1 Download and Installation

1. Go to FreeSurfer: Download (http://surfer.nmr.mgh.harvard.edu/fswiki/Download) and download the corresponding version for your system. In my case this was the *Linux CentOS 6 x86_64 (64b) stable v5.3.0* version. The file is called `freesurfer-Linux-centos6_x86_64-stable-pub-v5.3.0.tar.gz`.
2. Unpack FreeSurfer's binary folder to the place where you want the software to be at. In my case, I want to install FreeSurfer at `/usr/local/freesurfer`, which in my case needs root privilege. In my case this all can be done with the following command:

```
sudo tar xzvf \
    ~/Downloads/freesurfer-Linux-centos6_x86_64-stable-pub-v5.3.0.tar.gz
    -C /usr/local/
```

3. The usage of FreeSurfer requires a license file. Therefore, before you can use FreeSurfer, make sure to register (https://surfer.nmr.mgh.harvard.edu/registration.html). The content of the license file looks something like this:

```
username@gmail.com
12345
 *A3zKO68mtFu5
```

   This key has to be saved under a file with the name *.license* and has to be stored at your `$FREESURFER_HOME` location. In my case, this is `/usr/local/freesurfer`. To create this file in an Ubuntu environment use the following command:

```
sudo gedit /usr/local/freesurfer/.license
```

   Now copy the license code into this file, and save and close it.

4. The last thing you have to do before you can use FreeSurfer is to tell your system where the software package is. To do this, add the following code to your *.bashrc* file:

```
#FreeSurfer
export FREESURFER_HOME=/usr/local/freesurfer
source $FREESURFER_HOME/SetUpFreeSurfer.sh
```

5.2.2 Test FreeSurfer

After setting everything up, we can test if FreeSurfer is set up correctly and run a test with the following command:

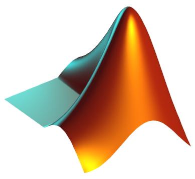
```
#Test 1
freeview -v $SUBJECTS_DIR/bert/mri/brainmask.mgz \
         -v $SUBJECTS_DIR/bert/mri/aseg.mgz:colormap=lut:opacity=0.2 \
         -f $SUBJECTS_DIR/bert/surf/lh.white:edgecolor=yellow \
         -f $SUBJECTS_DIR/bert/surf/rh.white:edgecolor=yellow \
         -f $SUBJECTS_DIR/bert/surf/lh.pial:annot=aparc:edgecolor=red \
         -f $SUBJECTS_DIR/bert/surf/rh.pial:annot=aparc:edgecolor=red

#Test 2
tksurfer bert lh pial -curv -annot aparc.a2009s.annot
```

Note: On a new Ubuntu System this might lead to the following error: `freeview.bin: error while loading shared libraries: libjpeg.so.62: cannot open shared object file: No such file or directory`. This is a common error on Ubuntu and can be solved with the following command:

```
cd /usr/lib/x86_64-linux-gnu
sudo ln -s libjpeg.so.8 libjpeg.so.62
sudo ln -s libtiff.so.4 libtiff.so.3
```

Alternately, this error can sometimes also be overcome by installing the libjpeg62-dev package with the following command: `sudo apt-get install libjpeg62-dev`

5.3 MATLAB



Nowadays almost all scientific fields take advantage of MATLAB (http://www.mathworks.com/). Neuroscience is no exception in this and also some of Nipype's recommended interfaces can (but don't have to) take advantage of MATLAB, e.g. SPM, FSL, FreeSurfer.

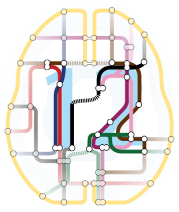
Having MATLAB is always a good thing, and as I myself rely often on algorithms from the SPM (http://www.fil.ion.ucl.ac.uk/spm/) interface, I need it to be on my system. A detailed documentation on how to install MATLAB can be found here (http://www.mathworks.com/help/). In my case, MATLAB is installed at the following location: `/usr/local/MATLAB/R2014a`.

The only thing you need to do to run MATLAB on your Ubuntu System is to add the following lines to your `.bashrc` file:

```
#MATLAB
export PATH=/usr/local/MATLAB/R2014a/bin:$PATH
export MATLABCMD=/usr/local/MATLAB/R2014a/bin/glnxa64/MATLAB
```

To test if everything is set up correctly. Open a new Terminal and type in the command: "matlab".

5.4 SPM12



SPM (http://www.fil.ion.ucl.ac.uk/spm/) stands for Statistical Parametric Mapping and is probably one of the most widely-used neuroimaging analysis software package worldwide. SPM is based on MATLAB and therefore needs it to be installed on your system. Luckily, the previous step just made that sure.

As of 1st October 2014, SPM released it's newest version SPM12 (http://www.fil.ion.ucl.ac.uk/spm/software/spm12/). The Release Notes (http://www.fil.ion.ucl.ac.uk/spm/software/spm12/SPM12_Release_Notes.pdf) mention some important updates and I therefore recommend to use SPM12 (http://www.fil.ion.ucl.ac.uk/spm/software/spm12/) instead of SPM8 (http://www.fil.ion.ucl.ac.uk/spm/software/spm8/). Nonetheless, Nipype has no issue with either SPM8 (http://www.fil.ion.ucl.ac.uk/spm/software/spm8/) or SPM12 (http://www.fil.ion.ucl.ac.uk/spm/software/spm12/). Therefore, you can install the version that you prefer.

Note: There is a standalone version of SPM available that doesn't need MATLAB, but so far it isn't recommended as a lot of additional toolboxes don't work with the standalone, yet. For more information go on the SPM wikipage (https://en.wikibooks.org/wiki/SPM/Standalone).

5.4.1 Download and Installation

To download and install the newest version SPM12 (http://www.fil.ion.ucl.ac.uk/spm/software/spm12/) do as follows:

1. Got to SPM12's Download and registration (http://www.fil.ion.ucl.ac.uk/spm/software/download/) page and fill out the form. Under **Select SPM version required**, chose SPM12 (or SPM8 if preferred) and download the zip file.

2. Now, unpack the zip file and copy the content to the recommended folder `/usr/local/MATLAB/R2014a/toolbox/` use the following code:

   ```
   sudo unzip ~/Downloads/spm12.zip -d /usr/local/MATLAB/R2014a/toolbox/
   ```

   Note: You don't have to put the spm12 folder into this folder, just make sure that you tell your system where to find it.

3. Now tell your system where it can find SPM12 by adding the following line to your `.bashrc` file:

   ```
   #SPM12
   export SPM_PATH=/usr/local/MATLAB/R2014a/toolbox/spm12/
   ```

4. Now, set up your MATLAB `startup.m` script so that MATLAB knows where SPM12 is stored at. If you've already installed FreeSurfer, than the `startup.m` file should be at `~/matlab/startup.m`. Otherwise create it and save it at this location. Now add the following code to this file:

   ```
   %-SPM12-
   spm_path = getenv('SPM_PATH');
   if spm_path,
   ```

```
    addpath(spm_path);
end
```

Note: There are some interesting ways on how you can change the default behaviors of your SPM.

- **Example 1:** By default, SPM uses only 64MB of memory during GLM estimation. This can be changed by changing the `defaults.stats.maxmem` parameter. Change this value to `2^29` and use 512MB or to `2^30` and use 1GB of memory during GLM estimation. Another option only available in SPM12 is to set `defaults.stats.resmem = true;`. Setting this parameter to true means that the temporary files during GLM estimation are kept in memory and not stored on disk (if value is set to false). For more information about increasing the speed of your SPM see the official Faster SPM (https://en.wikibooks.org/wiki/SPM/Faster_SPM) section.
- **Example 2**: One computational unimportant but nice parameter to change is `defaults.ui.colour = [0.141 0 0.848];`. Change it to the recommended value and see the nice color change in your SPM GUI.

How to change those values: SPM8 and SPM12 differ a bit in how those changes have to be implemented. In SPM8 you can change the default behavior by directly changing the parameters in the `spm_defaults.m` file, stored in the `spm8` folder. If you want to change default values in SPM12, you should create a new file called `spm_my_defaults.m`, store it in your `spm12` folder. The first line of your `spm_my_defaults.m` file has to be `global defaults`, followed by all the parameters you want to change, e.g. `defaults.ui.colour = [0.141 0 0.848];`

5.4.2 Test SPM12

To test if SPM12 is set up correctly, open MATLAB and type in the command `spm fmri`. This can also be achieved in one command: `matlab -r "spm fmri"`.

5.5 ANTs

ANTs (http://stnava.github.io/ANTs/) stands for Advanced Normalization Tools and is a great software package for registration, segmentation and normalization of MRI data. I highly recommend to use ANTs for the normalization of your data. **Side note**: ANTs can also be used to create a very cool looking average brain (template) out of a your own population of subjects.

There are two ways how you can set up ANTS on your own system:

The **first** way is very fast and simple. Just download the newest release of ANTs from their official github homepage (https://github.com/stnava/ANTs/releases). Decompress the downloaded files and store them somewhere on your system, e.g. under `/usr/local/antsbin`. After you've done that, just add the following line to your `.bashrc` file so that your system knows where to find the ANTs binaries:

```
#ANTs
export PATH=/usr/local/antsbin/bin:$PATH
export ANTSPATH=/usr/local/antsbin/bin/
```

The **second** way to get ANTs on your system takes a bit longer, but guarantees that you have the newest version of ANTs, specifically compiled for your system. Do as follows:

1. Download the data from the official homepage http://stnava.github.io/ANTs/. I chose the "Download TAR Ball" option.

2. Unpack the just downloaded files to a subfolder in your download folder (or wherever you want) with the following command:

   ```
   tar xzvf ~/Downloads/stnava-ANTs-b4eb279.tar.gz -C ~/Downloads
   ```

3. The installation of ANTs differs from other installation by the fact that the software first has to be compiled before it can run on your system. The code has to be compiled to create the binary files specific for your system. To do this, we first need to create a temporary folder to store all important files. This can bed one with the following code:

   ```
   mkdir ~/Downloads/stnava-ANTs-b4eb279/antsbin
   ```

4. Go into this folder with `cd ~/Downloads/stnava-ANTs-b4eb279/antsbin` and proceed with the following steps:

   ```
   #1. Install ccmake and other dependencies to be able to compile the code
   sudo apt-get install cmake-curses-gui build-essential zlib1g-dev

   #2.
   ccmake ../../stnava-ANTs-b4eb279

   #3. Press the [c] button to configure the compilation options

   #4. Change the CMAKE_INSTALL_PREFIX value to /usr/local/antsbin

   #5. First press the [c] and than the [g] button to generate the code

   #6. Now everything is set up to compile the code
   make -j 4

   #7. Now you're ready to install ANTs with the following commands:
   cd ANTS-build/
   sudo make install

   #8. Use the following command to copy important scripts from
   #   the ANTs folder 'stnava-ANTs-b4eb279/Scripts' into the folder
   #   where you've stored the ANTs binaries
   sudo cp ~/Downloads/stnava-ANTs-b4eb279/Scripts/* /usr/local/antsbin/bin/

   #9. Now that everything is done you can delete the temporary folder
   #   'stnava-ANTs-b4eb279' again.
   ```

5. Just one last thing before your can run ANTs, add the following lines to your `.bashrc` file:

   ```
   #ANTs
   export PATH=/usr/local/antsbin/bin:$PATH
   export ANTSPATH=/usr/local/antsbin/bin/
   ```

5.6 AFNI

AFNI (https://afni.nimh.nih.gov/afni/) is an open source software package specialized on the analysis of functional MRI. To see a list of all AFNI algorithms that can be used with Nipype go to interfaces.afni.preprocess (http://nipype.readthedocs.io/en/latest/interfaces/generated/nipype.interfaces.afni.preprocess.html).

If you've installed the NeuroDebian repository, just use the following command to install AFNI on your system: `sudo apt-get install afni`

To be able to run AFNI make sure to add the following lines of code to your `.bashrc` file:

```
#AFNI
export PATH=/usr/lib/afni/bin:$PATH
```

5.7 Additional interfaces

There are many additional interfaces, such as Camino (http://camino.cs.ucl.ac.uk/), MRtrix (http://www.brain.org.au/software/mrtrix/index.html), Slicer (http://slicer.org/), ConnectomeViewer (http://www.connectomics.org/viewer/), for which I haven't created an installation guide yet. This is also due to my lack of knowledge about them. Feel free to help me to complete this list.

CHAPTER

SIX

CLEAN UP YOUR SYSTEM

Now that everything is downloaded and installed, make sure that everything is correctly updated with the following command:

```
#Update and upgrade your system
sudo apt-get update && sudo apt-get upgrade

#Optional 1: Upgrade your distribution with
sudo apt-get dist-upgrade

#Optional 2: Clean your system and remove unused packages
sudo apt-get autoremove && sudo apt-get autoclean
sudo apt-get remove && sudo apt-get clean
```

CHAPTER

SEVEN

TEST YOUR SYSTEM

Nipype is installed, recommended interfaces are ready to go and so are you. But before you want to start your first steps with Nipype, I recommend you to test your system first. To do this open up an IPython environment (open a terminal and start IPython with the command `ipython`) and run the following code:

```
# Import the nipype module
import nipype

# Optional: Use the following lines to increase verbosity of output
nipype.config.set('logging', 'workflow_level',  'CRITICAL')
nipype.config.set('logging', 'interface_level', 'CRITICAL')
nipype.logging.update_logging(nipype.config)

# Run the test: Increase verbosity parameter for more info
nipype.test(doctests=False)
```

This test can take some minutes but if all goes well you will get an output more or less like this:

```
Ran 7454 tests in 71.160s

OK (SKIP=10)
Out[7]: <nose.result.TextTestResult run=7454 errors=0 failures=0>
```

Don't worry if some modules are being skipped or some side modules show up as errors or failures during the run. As long as no main modules cause any problems, you're fine. The number of tests and time will vary depending on which interfaces you have installed on your system. But if you receive an `OK`, `errors=0` and `failures=0` then everything is ready.

Congratulation! You now have a system with a fully working Nipype environment. Have fun!

Note: The first time I used MATLAB in Nipype I got the following error message:

```
Standard error:
MATLAB code threw an exception:
SPM not in matlab path
File:/home/username/workingdir/sliceTiming/pyscript_slicetiming.m
Name:pyscript_slicetiming
Line:6
Return code: 0
Interface MatlabCommand failed to run.
Interface SliceTiming failed to run.
```

As mentioned in the error message *SPM not in matlab path*, Nipype can't find the path to SPM. To change that, you can either add `addpath /usr/local/MATLAB/R2014a/toolbox/spm12b` to your `startup.m` file, stored at `~/matlab/startup.m` or add the following line of code at the beginning of your Nipype script:

```
from nipype.interfaces.matlab import MatlabCommand
MatlabCommand.set_default_paths('/usr/local/MATLAB/R2014a/toolbox/spm12b')
```

CHAPTER

EIGHT

PREPARE YOUR DATASET

We can't start to learn how to use Nipype before we don't have any data to work on. The fact that you are interested in using Nipype implies that you probably already have access to an MRI dataset. You are of course free to do your first steps with Nipype directly on your own dataset. But if you're new to the topic or want to be sure that you get the same results like I do, I recommend to use a tutorial dataset which we can find over on OpenfMRI (https://openfmri.org/).

8.1 Download the tutorial dataset

OpenfMRI (https://openfmri.org/) is an awesome new online database where you can upload ans share your fMRI dataset or download datasets from other researchers. Such an open approach to science allows you and other researchers to profit from each other and helps the field as a whole to progress faster. A list of all datasets available on OpenfMRI can be found on openfmri.org (https://openfmri.org/dataset/).

8.1.1 Download the dataset from OpenfMRI

I chose to use the dataset DS102: Flanker task (event-related) (https://openfmri.org/dataset/ds000102) as the tutorial dataset for this beginner's guide because the dataset is complete, in good quality, the experiment is representative for most fMRI experiments and the dataset as a whole is with its 1.8GB not too big to download. To download the dataset go to the button of the DS102: Flanker task (event-related) (https://openfmri.org/dataset/ds000102) page and click on the link: Raw data on AWS (https://openfmri.s3.amazonaws.com/tarballs/ds102_raw.tgz).

Note: If you are on a Linux system, you can also download the tutorial dataset directly to your `Downloads` folder with the following command:

```
wget https://openfmri.s3.amazonaws.com/tarballs/ds102_raw.tgz ~/Downloads
```

8.1.2 Unpack and prepare the folder structure

The file you've just downloaded contains amongst others one anatomical and two functional MRI scans and the behavioral response and onsets of stimuli during those functional scans. The whole dataset consists of 26 subjects. Information about gender and age of each subject is stored in the *demographics.txt* file, also found in downloaded file. To reduce the total time of computation, this tutorial will only analyze the first 10 subjects of the whole dataset. Now let's get ready.

What we want is an experiment folder called `nipype_tutorial` that contains a folder called `data` where we save the raw data of the first 10 subjects into. In the end, the structure should be something like this

```
nipype_tutorial
|-- data
    |-- demographics.txt
    |-- sub001
    |   |-- behavdata_run001.txt
    |   |-- behavdata_run002.txt
    |   |-- onset_run001_cond001.txt
    |   |-- onset_run001_cond002.txt
    |   |-- onset_run001_cond003.txt
    |   |-- onset_run001_cond004.txt
    |   |-- onset_run002_cond001.txt
    |   |-- onset_run002_cond002.txt
    |   |-- onset_run002_cond003.txt
    |   |-- onset_run002_cond004.txt
    |   |-- run001.nii.gz
    |   |-- run002.nii.gz
    |   |-- struct.nii.gz
    |-- sub0..
    |-- sub010
        |-- behav...
        |-- onset_...
        |-- run...
        |-- struct.nii.gz
```

This can either be done manually or with the following code:

```
# Specify important variables
ZIP_FILE=~/Downloads/ds102_raw.tgz   #location of download file
TUTORIAL_DIR=~/nipype_tutorial       #location of experiment folder
TMP_DIR=$TUTORIAL_DIR/tmp            #location of temporary folder
DATA_DIR=$TUTORIAL_DIR/data          #location of data folder

# Unzip ds102 dataset into TMP_DIR
mkdir -p $TMP_DIR
tar -zxvf $ZIP_FILE -C $TMP_DIR

# Copy data of first ten subjects into DATA_DIR
for id in $(seq -w 1 10)
do
    echo "Creating dataset for subject: sub0$id"
    mkdir -p $DATA_DIR/sub0$id
    cp $TMP_DIR/ds102/sub0$id/anatomy/highres001.nii.gz \
       $DATA_DIR/sub0$id/struct.nii.gz

    for session in run001 run002
    do
        cp $TMP_DIR/ds102/sub0$id/BOLD/task001_$session/bold.nii.gz \
           $DATA_DIR/sub0$id/$session.nii.gz
        cp $TMP_DIR/ds102/sub0$id/behav/task001_$session/behavdata.txt \
           $DATA_DIR/sub0$id/behavdata_$session.txt

        for con_id in {1..4}
        do
            cp $TMP_DIR/ds102/sub0$id/model/model001/onsets/task001_$session/cond00
↪$con_id.txt \
               $DATA_DIR/sub0$id/onset_${session}_cond00$con_id.txt
        done
    done

    echo "sub0$id done."
done

# Copy information about demographics, conditions and tasks into DATA_DIR
cp $TMP_DIR/ds102/demographics.txt $DATA_DIR/demographics.txt
cp $TMP_DIR/ds102/models/model001/* $DATA_DIR/.

# Delete the temporary folder
rm -rf $TMP_DIR
```

Hint: You can download this code as a script here: tutorial_1_create_dataset.sh (https://github.com/miykael/nipype-beginner-s-guide/blob/master/scripts/tutorial_1_create_dataset.sh)

8.1.3 Acquire scan and experiment parameters

One of the most important things when analyzing any data is to know your data. What does it consist of, how was it recorded, what are its characteristics and most importantly, what does it look like. The DS102: Flanker task (event-related) (https://openfmri.org/dataset/ds000102) page helps us already to answer many of those parameters:

Scan parameters

- Magnetic field: 3T [Tesla]
- Head coil: Siemens standard, which probably means 32-channel
- **Information about functional acquisition**
  - Acquisition type: contiguous echo planar imaging (EPI)
  - Number of volumes per session: 146
  - Number of slices per volume: 40
  - Slice order: unknown
  - Repetition time (TR): 2000ms
  - Field of view (FOV): 64x64
  - Voxel size: 3x3x4mm
- **Information about anatomical acquisition**
  - Acquisition type: magnetization prepared gradient echo sequence (MPRAGE)
  - Number of slices: 176
  - Repetition time (TR): 2500ms
  - Field of view (FOV): 256mm

Experiment parameters

- **Task**: On each trial, participants used one of two buttons on a response pad to indicate the direction of a central arrow in an array of 5 arrows. In *congruent* trials the flanking arrows pointed in the same direction as the central arrow (e.g., < < < < <), while in more demanding *incongruent* trials the flanking arrows pointed in the opposite direction (e.g., < < > < <).
- **Condition**: congruent and incongruent
- **Scan sessions**: Subjects performed two 5-minute blocks, each containing 12 congruent and 12 incongruent trials, presented in a pseudo-random order.

So what do we know?

- We know that we have two functional scans per subjects, in our case called *run001* and *run002*. Each functional scan represents 5min of scan time and consists of 146 volumes. Each of those volume consists 40 slices (with a thickness of 4mm) and each slice consists of 64x64 voxel with the size of 3x3mm. The TR of each volume is 2000ms.
- So far we don't know what the slice order of the functional acquisition is. But a closer look at the provided references tells us that the data was acquired with an *interleaved* slice to slice order. The fact that it is not stated if it is ascending or descending interleaved means that the acquisition is most certainly ascending.
- We know that we have one anatomical scan per subject, in our case called *struct* and that this anatomical scan consists of 176 slices with each having a FOV of 256mm. This hints to an isometric voxel resolution of 1x1x1mm.
- We know the task of the experiment and that it consists of two conditions, *congruent* and *incongruent*. With each condition being presented 12 times per session.
- We know from the onset file found in the subject folder what the actual onset of the two conditions are, but we have no clear information about the duration of each stimulation. The description of the design on OpenfMRI.org (https://openfmri.org/dataset/ds000102) tells us that the inter-trial interval varies between 8 and 14s with a mean of 12s. A closer look at the references tells us that the subjects first see a fixation cross for 500ms, followed by the congruent or incongruent condition (i.e. the arrows) for 1500ms, followed by a blank screen for 8000-12000ms.

8.1.4 Check the data

It is always important to look at your data and verify that it actually is recorded the way it should be. To take a look at the data, I usually use FreeSurfer's `freeview`. For example, if you want to load the anatomical scan of all ten subjects use the following code:

```
freeview -v ~/nipype_tutorial/data/sub00*/struct.nii.gz
```

To verify the information about the scan parameters we can use `fslinfo`. `fslinfo` allows us to read the header information of a NIfTI file and therefore get information about voxel resolution and TR. For example, reading the header of the anatomical scan of subject2 with the command `fslinfo ~/nipype_tutorial/data/sub002/struct.nii.gz` gives us following output:

```
data_type      INT16
dim1           176
dim2           256
dim3           256
dim4           1
datatype       4
```

```
pixdim1         1.000000
pixdim2         1.000000
pixdim3         1.000000
pixdim4         0.000000
cal_max         0.0000
cal_min         0.0000
file_type       NIFTI-1+
```

This output tells us, that the anatomical volume consists of 256x256x176 voxels with each having 1x1x1mm resolution.

Using the same command on a functional scan of subject 2 gives as following output:

```
data_type       INT16
dim1            64
dim2            64
dim3            40
dim4            146
datatype        4
pixdim1         3.000000
pixdim2         3.000000
pixdim3         4.000000
pixdim4         2000.000000
cal_max         0.0000
cal_min         0.0000
file_type       NIFTI-1+
```

This output tells us that this functional scan consists of 146 volumes, of which each consists of 64x64x40 voxels with a resolution of 3x3x4mm. *pixdim4* gives us additionally information about the TR of this functional scan.

Note: Just as a side note: The *data_type* of a NIfTI file tells you the amount of bits used to store the value of each voxel. **INT16** stands for *signed short* (16 bits/voxel), **INT32** stands for *signed int* (32 bits/voxel) and **FLOAT64** stands for *float* (64 bits/voxel). The more bits used for storing a voxel value the bigger the whole NIfTI file.

If you want to change the data type of your image, either use Nipype's function `ChangeDataType` found in the `nipype.interfaces.fsl.maths` package (read more here (http://nipype.readthedocs.io/en/latest/interfaces/generated/nipype.interfaces.fsl.maths.html#changedatatype)) or use `fslmaths` directly to change the data_type to INT32 with the following command:

```
fslmaths input.nii output.nii -odt int
```

8.1.5 For those who use their own dataset

If you want to use your own dataset, make sure that you know the following parameters:

- Number of volumes, number of slices per volume, slice order and TR of the functional scan.
- Number of conditions during a session, as well as onset and duration of stimulation during each condition.

Important: Make sure that the layout of your data is similar to the one stated above, so that further code is also applicable for your case.

8.2 Make the dataset ready for Nipype

8.2.1 Convert your data into NIfTI format

You don't have to do this step if you're using the tutorial dataset. But chances are that you soon want to analyze your own recorded dataset. And most often, the images coming directly from the scanner are not in the common `NIfTI` format, but rather in a scanner specific format (e.g. `DICOM`, `PAR/REC`, etc.). This means you first have to convert your data from this specific scanner format to the standard NIfTI format.

Probably the most common scanner format is DICOM. Therefore, the following section will cover how you can convert your files from DICOM to NIfTI. There are many different tools that you can use to convert your files. For example, if you like to have a nice GUI to convert your files, use MRICron (http://www.mccauslandcenter.sc.edu/mricro/mricron/)'s MRIConvert (http://lcni.uoregon.edu/~jolinda/MRIConvert/). But for this Beginner's Guide we will use FreeSurfer's `mri_convert` function, as it is rather easy to use and doesn't require many steps.

But first, as always, be aware of your folder structure. So let's assume that we've stored our dicoms in a folder called `raw_data` and that the folder structure looks something like this:

```
raw_dicom
|-- sub001
|    |-- t1w_3d_MPRAGE
|    |    |-- 00001.dcm
|    |    |-- ...
|    |    |-- 00176.dcm
|    |-- fmri_run1_long
|    |    |-- 00001.dcm
|    |    |-- ...
|    |    |-- 00240.dcm
|    |-- fmri_run2_long
|         |-- ...
|-- sub0..
|-- sub010
```

This means, that we have one folder per subject with each containing another folder, one for the structural T1 weighted image and 2 for the functional T2 weighted images. The conversion of the dicom files in those folders is rather easy. If you use FreeSurfer's `mri_convert` function, the command is as as follows: `mri_convert <in volume> <out volume>`. You have to replace `<in volume>` by the actual path to any one dicom file in the folder and `<out volume>` with the name for your outputfile.

So, to accomplish this with some few terminal command, we first have to tell the system the path and names of the folders that we later want to feed to the `mri_convert` function. This is done by the following variables (line 1 to 6). If this is done, we only have to run the loop (line 8 to 17) to actually run `mri_convert` for each subject and each scanner image.

```
1  TUTORIAL_DIR=~/nipype_tutorial     # location of experiment folder
2  RAW_DIR=$TUTORIAL_DIR/raw_dicom    # location of raw data folder
3  T1_FOLDER=t1w_3d_MPRAGE            # dicom folder containing anatomical scan
4  FUNC_FOLDER1=fmri_run1_long        # dicom folder containing 1st     functional scan
5  FUNC_FOLDER2=fmri_run2_long        # dicom folder containing 2nd functional scan
6  DATA_DIR=$TUTORIAL_DIR/data        # location of output folder
7
8  for id in $(seq -w 1 10)
9  do
10     mkdir -p $DATA_DIR/sub0$id
11     mri_convert $RAW_DIR/sub0$id/$T1_FOLDER/00001.dcm    $DATA_DIR/sub0$id/struct.nii.
 ↪gz
```

```
    mri_convert $RAW_DIR/sub0$id/$FUNC_FOLDER1/00001.dcm $DATA_DIR/sub0$id/run001.nii.gz
    mri_convert $RAW_DIR/sub0$id/$FUNC_FOLDER2/00001.dcm $DATA_DIR/sub0$id/run002.nii.gz
done
```

8.2.2 Run FreeSurfer's recon-all

Not mandatory but highly recommended is to run FreeSurfer's `recon-all` process on the anatomical scans of your subject. `recon-all` is FreeSurfer's cortical reconstruction process that automatically creates a parcellation of cortical (https://surfer.nmr.mgh.harvard.edu/fswiki/CorticalParcellation) and a segmentation of subcortical (http://freesurfer.net/fswiki/SubcorticalSegmentation) regions. A more detailed description about the `recon-all` process can be found on the official homepage (http://surfer.nmr.mgh.harvard.edu/fswiki/recon-all).

As I said, you don't have to use FreeSurfer's `recon-all` process, but you want to! Because many of FreeSurfer's other algorithms require the output of `recon-all`. The only negative point about `recon-all` is that it takes rather long to process a single subject. My average times are between 12-24h, but it is also possible that the process takes up to 40h. All of it depends on the system you are using. So far, `recon-all` can't be run in parallel. Luckily, if you have an 8 core processor with enough memory, you should be able to process 8 subjects in parallel.

Run recon-all on the tutorial dataset (terminal version)

The code to run `recon-all` on a single subject is rather simple, i.e. `recon-all -all -subjid sub001`. The only thing that you need to keep in mind is to tell your system the path to the freesurfer folder by specifying the variable `SUBJECTS_DIR` and that each subject you want to run the process on has a according anatomical scan in this freesurfer folder under `SUBJECTS_DIR`.

To run `recon-all` on the 10 subjects of the tutorial dataset you can run the following code:

```
1  # Specify important variables
2  export TUTORIAL_DIR=~/nipype_tutorial         #location of experiment folder
3  export DATA_DIR=$TUTORIAL_DIR/data            #location of data folder
4  export SUBJECTS_DIR=$TUTORIAL_DIR/freesurfer  #location of freesurfer folder
5
6  for id in $(seq -w 1 10)
7  do
8      echo "working on sub0$id"
9      mkdir -p $SUBJECTS_DIR/sub0$id/mri/orig
10     mri_convert $DATA_DIR/sub0$id/struct.nii.gz \
11                 $SUBJECTS_DIR/sub0$id/mri/orig/001.mgz
12     recon-all -all -subjid sub0$id
13     echo "sub0$id finished"
14 done
```

This code will run the subjects in sequential order. If you want to process the 10 subjects in (manual) parallel order, delete line 12 - `recon-all -all -subjid sub0$id` - from the code above, run it and than run the following code, each line in its own terminal:

```
export SUBJECTS_DIR=~/nipype_tutorial/freesurfer; recon-all -all -subjid sub001
export SUBJECTS_DIR=~/nipype_tutorial/freesurfer; recon-all -all -subjid sub002
...
export SUBJECTS_DIR=~/nipype_tutorial/freesurfer; recon-all -all -subjid sub010
```

Note: If your MRI data was recorded on a 3T scanner, I highly recommend to use the `-nuintensitycor-3T` flag on the `recon-all` command, e.g. `recon-all -all -subjid sub0$id -nuintensitycor-3T`. This flag was created specifically for 3T scans and improves the brain segmentation accuracy by optimizing non-uniformity correction using N3 (http://web.mysites.ntu.edu.sg/zvitali/publications/documents/N3_NI.pdf).

Hint: You can download this code as a script here: tutorial_2_recon_shell.sh (https://github.com/miykael/nipype-beginner-s-guide/blob/master/scripts/tutorial_2_recon_shell.sh)

Run recon-all on the tutorial dataset (Nipype version)

If you run `recon-all` only by itself, I recommend you to use the terminal version shown above. But of course, you can also create a pipeline and use Nipype to do the same steps. This might be better if you want to make better use of the parallelization implemented in Nipype or if you want to put `recon-all` in a bigger workflow.

I won't explain to much how this workflow actually works, as the structure and creation of a common pipeline is covered in more detail in the next section. But to use Nipype to run FreeSurfer's `recon-all` process do as follows:

```python
# Import modules
import os
from os.path import join as opj
from nipype.interfaces.freesurfer import ReconAll
from nipype.interfaces.utility import IdentityInterface
from nipype.pipeline.engine import Workflow, Node

# Specify important variables
experiment_dir = '~/nipype_tutorial'             # location of experiment folder
data_dir = opj(experiment_dir, 'data')  # location of data folder
fs_folder = opj(experiment_dir, 'freesurfer')  # location of freesurfer folder
subject_list = ['sub001', 'sub002', 'sub003',
                'sub004', 'sub005', 'sub006',
                'sub007', 'sub008', 'sub009',
                'sub010']                        # subject identifier
T1_identifier = 'struct.nii.gz'                  # Name of T1-weighted image

# Create the output folder - FreeSurfer can only run if this folder exists
os.system('mkdir -p %s'%fs_folder)

# Create the pipeline that runs the recon-all command
reconflow = Workflow(name="reconflow")
reconflow.base_dir = opj(experiment_dir, 'workingdir_reconflow')

# Some magical stuff happens here (not important for now)
infosource = Node(IdentityInterface(fields=['subject_id']),
                  name="infosource")
infosource.iterables = ('subject_id', subject_list)

# This node represents the actual recon-all command
reconall = Node(ReconAll(directive='all',
                         #flags='-nuintensitycor-3T',
                         subjects_dir=fs_folder),
                name="reconall")

# This function returns for each subject the path to struct.nii.gz
```

```python
def pathfinder(subject, foldername, filename):
    from os.path import join as opj
    struct_path = opj(foldername, subject, filename)
    return struct_path

# This section connects all the nodes of the pipeline to each other
reconflow.connect([(infosource, reconall, [('subject_id', 'subject_id')]),
                   (infosource, reconall, [(('subject_id', pathfinder,
                                             data_dir, T1_identifier),
                                            'T1_files')]),
                   ])

# This command runs the recon-all pipeline in parallel (using 8 cores)
reconflow.run('MultiProc', plugin_args={'n_procs': 8})
```

After this script has run, all important outputs will be stored directly under `~/nipype_tutorial/freesurfer`. But the running of the `reconflow` pipeline also created some temporary files. As defined by the script above, those files were stored under `~/nipype_tutorial/workingdir_reconflow`. Now that the script has run you can delete this folder again. Either do this manually, use the shell command `rm -rf ~/nipype_tutorial/workingdir_reconflow` or add the following lines to the end of the python script above:

```python
# Delete all temporary files stored under the 'workingdir_reconflow' folder
os.system('rm -rf %s'%reconflow.base_dir)
```

Note: In the code above, if we don't create the `freesurfer` output folder on line 19, we would get following error:

```
TraitError: The 'subjects_dir' trait of a ReconAllInputSpec instance must be an␣
↪existing
directory name, but a value of '~/nipype_tutorial/freesurfer' <type 'str'> was␣
↪specified.
```

Also, if your data was recorded on a 3T scanner and you want to use the mentioned `-nuintensitycor-3T` flag, just uncomment line 32, i.e. delete the `#` sign before `flags='-nuintensitycor-3T'` on line 32.

Hint: You can download this code as a script here: tutorial_2_recon_python.py (https://github.com/miykael/nipype-beginner-s-guide/blob/master/scripts/tutorial_2_recon_python.py)

8.3 Resulting Folder Structure

After we've prepared our data and run the `recon-all` process the folder structure of our experiment folder should look as follows:

```
nipype_tutorial
|-- rawdata (optional)
|-- data
|    |-- sub001
|    |-- sub0..
|    |-- sub010
|-- freesurfer
     |-- sub001
```

```
    |-- sub0..
    |-- sub010
```

CHAPTER

NINE

HOW TO BUILD A PIPELINE

So, you've installed Nipype (http://miykael.github.io/nipype-beginner-s-guide/installation.html) on your system? And you've prepared your dataset (http://miykael.github.io/nipype-beginner-s-guide/prepareData.html) for the analysis? This means that you are ready to start this tutorial.

The following section is a general step by step introduction on how to build a pipeline. It will first introduce you to the building blocks of any pipeline, then show you an example of how a basic pipeline is implemented, how to read the output, and most importantly how to tackle problems. At the end you should understand what a pipeline is, how the key parts interact with each other, and how to solve certain issues. In short, you should be able to build any kind of neuroimaging pipeline that you like. So let's get started!

9.1 Important building blocks

Before we get into the details, we should explain that Nipype pipelines, or workflows, the terms are synonomous, are made up of *nodes* and *workflows*. We will generally use 'workflow' when referring to section in a script and 'pipeline' to refer to the whole construction that is actually run.

Briefly, you can think of a *node* as a unit of processing; for example, a node will run the program to smooth data. Nodes are most often associated with analytic interfaces to, say, FSL, SPM, or AFNI, but they can also be a function to gather the list of image files to be used. Each node is a step in the pipeline.

A *workflow* is a collection of nodes and the rules that connect the nodes to each other. Typically, the output of one node is attached to the input of another node in a workflow. Workflows can also connect other workflows, so you might have a preprocessing workflow, a first-level workflow, a second-level workflow, etc., perhaps with other nodes in between.

Refer back to the diagram of a Nipype workflow on the Nipype and Neuroimaging (http://miykael.github.io/nipype-beginner-s-guide/nipypeAndNeuroimaging.html#nipype-workflow) page. There, the purple box represents an fMRI workflow, and it contains a Preprocessing workflow (the pink box), a functional analysis workflow (the light green box), which in turn contains a 1st level analysis workflow and a 2nd level analysis workflow.

9.2 Example preprocessing pipeline

The following example pipeline is based on the preprocessing steps of a functional MRI study. We can usefully categorize in a general way the sections of a pipeline script based on what function each section plays. The sections are common to virtually all pipelines, even though the specific nodes may vary. For that reason, it's important to understand the sections, how they relate to each other, and what their role in the overall pipeline is.

- **Import modules**: The first step in any script is to import necessary functions and modules needed for data manipulation and analysis.

- **Specify variables**: The second step is to define all the variables you will use throughout the script. This is best done in just one place, and I recommended you to do this as the second step, right after module importation.
- **Specify Nodes**: The third step is to define the nodes you will use in

your pipeline. Each node will assign a name to its inputs and to its outputs, and any parameters that are needed for the node to do its job will be defined or assigned to an argument here.

- **Specify Workflows & Connect Nodes**: The fourth step is to define at least

one workflow, though possibly more. The workflows order the nodes and connect them to each other to create the processing stream, insuring that the nodes are executed in the proper order and produce the necessary input for the next node in the workflow.

- **Input & Output Stream**: The fifth step is to specify the structure of

your data folders. The purpose of the input stream is to specify in which folders and under what names the input data is stored. The output stream specifies in which folders output data is stored, which may well be different from where the input data is stored.

- **Run Workflow**: The final step is to actually run the outermost

workflow to do the work and make you famous!

So, make yourself ready, start an IPython environment (type `ipython` into the terminal) and have fun creating this example workflow.

9.2.1 Import modules

The first thing you should do in any script is to import the modules you want to use in your script. In our case, most modules are actually interfaces to other software packages (e.g. FSL, FreeSurfer or SPM) and can be found in the `nipype` module.

If you want to import a module as a whole use the command structure `import <module> as <name>`. For example:

```python
# Import the SPM interface under the name spm
import nipype.interfaces.spm as spm
```

If you only want to import a certain function of a module use the command structure `from <module> import <function>`. For example:

```python
# Import the function 'maths' from the FSL interface
from nipype.interfaces.fsl import maths
```

And if you want to import multiple functions or classes from a module use the following structure:

```python
# Import nipype Workflow, Node and MapNode objects
from nipype.pipeline.engine import Workflow, Node, MapNode
```

9.2.2 Specify variables

There are always variables that change between analysis or that are specific for a certain computer structure. That's why it is important to keep them all together and at one place. This allows you to be fast, flexible and to keep the changes in your script just to this section.

So which variables should you declare in this section? **All of them!** Every variable that you change more than once between analysis should be specified here.

For example:

```
# What is the location of your experiment folder
experiment_dir = '~/nipype_tutorial'

# What are the names of your subjects
subjects = ['sub001','sub002','sub003']

# What is the name of your working directory and output folder
output_dir =  'output_firstSteps'
working_dir = 'workingdir_firstSteps'

# What are experiment specific parameters
number_of_slices = 40
time_repetition = 2.0
fwhm = 8
```

9.2.3 Specify Nodes

It is impossible to build a pipeline without any scaffold or objects to build with. Therefore, we first have to create those scaffolds (i.e. `workflows`) and objects (i.e. `nodes` or other `workflows`).

Nodes

A node is an object that represents a certain interface function, for example SPM's `Realign` method. Every node has always at least one input and one output field. The existent of those fields allow Nipype to connect different nodes to each other and therefore guide the stream of input and output between the nodes.

Input and Output Fields

Nipype provides so many different interfaces with each having a lot of different functions (for a list of all interfaces go here (http://nipy.org/nipype/interfaces/index.html). So how do you know which input and output field a given node has? Don't worry. There's an easy way how you can figure out which input fields are **mandatory** or **optional** and which output fields you can use.

Let's assume that we want to know more about FSL's function `SmoothEstimate`. First, make sure that you've imported the fsl module with the following python command `import nipype.interfaces.fsl as fsl`.

Now that we have access to FSL, we simply can run `fsl.SmoothEstimate.help()`. This will give us the following output:

```
Wraps command **smoothest**

Estimates the smoothness of an image

Examples
--------

>>> est = SmoothEstimate()
>>> est.inputs.zstat_file = 'zstat1.nii.gz'
>>> est.inputs.mask_file = 'mask.nii'
>>> est.cmdline
'smoothest --mask=mask.nii --zstat=zstat1.nii.gz'

```

```
14  Inputs::
15
16      [Mandatory]
17      dof: (an integer)
18          number of degrees of freedom
19          flag: --dof=%d
20          mutually_exclusive: zstat_file
21      mask_file: (an existing file name)
22          brain mask volume
23          flag: --mask=%s
24
25      [Optional]
26      args: (a string)
27          Additional parameters to the command
28          flag: %s
29      environ: (a dictionary with keys which are a value of type 'str' and
30           with values which are a value of type 'str', nipype default value: {})
31          Environment variables
32      ignore_exception: (a boolean, nipype default value: False)
33          Print an error message instead of throwing an exception in case the
34          interface fails to run
35      output_type: ('NIFTI_PAIR' or 'NIFTI_PAIR_GZ' or 'NIFTI_GZ' or
36           'NIFTI')
37          FSL output type
38      residual_fit_file: (an existing file name)
39          residual-fit image file
40          flag: --res=%s
41          requires: dof
42      zstat_file: (an existing file name)
43          zstat image file
44          flag: --zstat=%s
45          mutually_exclusive: dof
46
47  Outputs::
48
49      dlh: (a float)
50          smoothness estimate sqrt(det(Lambda))
51      resels: (a float)
52          number of resels
53      volume: (an integer)
54          number of voxels in mask
```

The first few lines *(line 1-3)* give as a short explanation of the function, followed by a short example on how to implement the function *(line 5-12)*. After the example come information about `Inputs` *(line 14-45)* and `Outputs` *(line 47-54)*. There are always some inputs that are **mandatory** and some that are **optional**. Which is not the case for outputs, as they are always optional. It's important to note that some of the inputs are mutually exclusive *(see line 19)*, which means that if one input is specified, another one can't be set and will result in an error if it is defined nonetheless.

Important: If you only want to see the **example** part of the information view, without the details about the input and output fields, use the command `fsl.SmoothEstimate?`.

If you want to find the location of the actual Nipype script that serves as an interface to the external software package, use also the command `fsl.SmoothEstimate?` and check out the 3rd line, called **File:**.

Note: If you want to brows through the different functions, or just want to view the help information in a nicer way, go to the official homepage and either navigate to Interfaces and Algorithms (http://nipy.org/nipype/interfaces/index.html) or Documentation (http://nipy.org/nipype/documentation.html).

Default value of Inputs

As you've might seen in the example above *(line 32)*, some input fields have Nipype specific default values. To figure out which default values are used for which functions, use the method `input_spec()`.

For example, if you want to know that the default values for SPM's Threshold function are, use the following command:

```
import nipype.interfaces.spm as spm
spm.Threshold.input_spec()
```

This will give you the following output:

```
contrast_index = <undefined>
extent_fdr_p_threshold = 0.05
extent_threshold = 0
force_activation = False
height_threshold = 0.05
height_threshold_type = p-value
ignore_exception = False
matlab_cmd = <undefined>
mfile = True
paths = <undefined>
spm_mat_file = <undefined>
stat_image = <undefined>
use_fwe_correction = True
use_mcr = <undefined>
use_topo_fdr = True
use_v8struct = True
```

Stand-alone nodes

Nodes are most of the time used inside a pipeline. But it is also possible to use one just by itself. Such a "stand-alone" node is often times very convenient when you run a python script and want to use just one function of a given dependency package, e.g. FSL, and are not really interested in creating an elaborate workflow.

Such a "stand-alone" node is also a good opportunity to introduce the implementation of nodes. Because there are many ways how you can create a node. Let's assume that you want to create a single node that runs FSL's Brain Extraction Tool function `BET` on the anatomical scan of our tutorial subject `sub001`. This can be achieved in the following three ways:

```
# First, make sure to import the FSL interface
import nipype.interfaces.fsl as fsl

# Method 1: specify parameters during node creation
mybet = fsl.BET(in_file='~/nipype_tutorial/data/sub001/struct.nii.gz',
                out_file='~/nipype_tutorial/data/sub001/struct_bet.nii.gz')
mybet.run()

```

```python
# Method 2: specify parameters after node creation
mybet = fsl.BET()
mybet.inputs.in_file = '~/nipype_tutorial/data/sub001/struct.nii.gz'
mybet.inputs.out_file = '~/nipype_tutorial/data/sub001/struct_bet.nii.gz'
mybet.run()

# Method 3: specify parameters when the node is executed
mybet = fsl.BET()
mybet.run(in_file='~/nipype_tutorial/data/sub001/struct.nii.gz',
          out_file='~/nipype_tutorial/data/sub001/struct_bet.nii.gz')
```

Hint: To check the result of this execution, run the following command in your terminal:

```
freeview -v ~/nipype_tutorial/data/sub001/struct.nii.gz \
            ~/nipype_tutorial/data/sub001/struct_bet.nii.gz:colormap=jet
```

Workflow Nodes

Most of the times when you create a node you want to use it later on in a workflow. The creation of such a "workflow" node is only partly different from the creation of "stand-alone" nodes. The implementation of a "workflow" node has always the following structure:

```python
nodename = Node(interface_function(), name='label')
```

- **nodename**: This is the name of the object that will be created.
- **Node**: This is the type of the object that will be created. In this case it is a `Node`. It can also be defined as a `MapNode` or a `Workflow`.
- **interface_function**: This is the name of the function this node should represent. Most of the times this function name is preceded by an interface name, e.g. `fsl.BET`.
- **label**: This is the name, that this node uses to create its working directory or to label itself in the visualized graph.

For example: If you want to create a node called `realign` that runs SPM's `Realign` function on a functional data `func.nii`, use the following code:

```python
# Make sure to import required modules
import nipype.interfaces.spm as spm     # import spm
import nipype.pipeline.engine as pe     # import pypeline engine

# Create a realign node - Method 1: Specify inputs during node creation
realign = pe.Node(spm.Realign(in_files='~/nipype_tutorial/data/sub001/func.nii'),
               name='realignnode')

# Create a realign node - Method 2: Specify inputs after node creation
realign = pe.Node(spm.Realign(), name='realignnode')
realign.inputs.in_files='~/nipype_tutorial/data/sub001/func.nii'

# Specify the working directory of this node (only needed for this specific example)
realign.base_dir = '~/nipype_tutorial/tmp'

# Execute the node
realign.run()
```

Note: This will create a working directory at `~/nipype_tutorial/tmp`, containing one folder called `realignnode` containing all temporary and final output files of SPM's `realign` function.

If you're curious what the realign function just calculated, use the following python commands to plot the estimated translation and rotation parameters of the functional scan `func.nii`:

```python
# Import necessary modules
import numpy
import matplotlib.pyplot as plt

# Load the estimated parameters
movement=numpy.loadtxt('~/nipype_tutorial/tmp/realignnode/rp_func.txt')

# Create the plots with matplotlib
plt.subplot(211)
plt.title('translation')
plt.plot(movement[:,:3])

plt.subplot(212)
plt.title('rotation')
plt.plot(movement[:,3:])

plt.show()
```

Iterables

Iterables are a special kind of input fields and any input field of any Node can be turned into an Iterable. Iterables are very important for the repeated execution of a workflow with slightly changed parameters.

For example, let's assume that you have a preprocessing pipeline and on one step smooth the data with a FWHM smoothing kernel of 6mm. But because you're not sure if the FWHM value is right you would want to execute the workflow again with 4mm and 8mm. Instead of running your workflow three times with slightly different parameters you could also just define the FWHM input field as an iterables:

```python
# Import necessary modules
import nipype.interfaces.spm as spm     # import spm
import nipype.pipeline.engine as pe     # import pypeline engine

# Create a smoothing node - normal method
smooth = pe.Node(spm.Smooth(), name = "smooth")
smooth.inputs.fwhm = 6

# Create a smoothing node - iterable method
smooth = pe.Node(spm.Smooth(), name = "smooth")
smooth.iterables = ("fwhm", [4, 6, 8])
```

The usage of Iterables causes the execution workflow to be splitted into as many different clones of itself as needed. In this case, three execution workflows would be created, where only the FWHM smoothing kernel would be different. The advantage of this is that all three workflows can be executed in parallel.

Usually, Iterables are used to feed the different subject names into the workflow, causing your workflow to create as many execution workflows as subjects. And depending on your system, all of those workflows could be executed in parallel.

For a more detailed explanation of Iterables go to the Iterables section (http://nipy.org/nipype/users/mapnode_and_iterables.html#iterables) on the official homepage.

MapNodes and Iterfields

A MapNode is a sub-class of a Node. It therefore has exactly the same properties as a normal Node. The only difference is that a MapNode can put multiple input parameters into one input field, where a normal Node only can take one.

The creation of a MapNode is only slightly different to the creation of a normal Node.

```
nodename = MapNode(interface_function(), name='label', iterfield=['in_file'])
```

First, you have to use `MapNode` instead of `Node`. Second, you also have to define which of the input fields can receive multiple parameters at once. An input field with this special properties is also called an `iterfield`. For a more detailed explanation go to MapNode and iterfield (http://nipy.org/nipype/users/mapnode_and_iterables.html#mapnode-and-iterfield) on the official homepage.

Individual Nodes

There are situations where you need to create your own Node that is independent from any other interface or function provided by Nipype. You need a Node with your specific input and output fields, that does what you want. Well, this can be achieved with Nipype's `Function` function from the `utility` interface.

Let's assume that you want to have a node that takes as an input a NIfTI file and returns the voxel dimension and the TR of this file. We will read the voxel dimension and the TR value of the NIfTI file with `nibabel`'s `get_header()` function.

Here is how it's done:

```
# Import necessary modules
import nipype.pipeline.engine as pe
from nipype.interfaces.utility import Function

# Define the function that returns the voxel dimension and TR of the in_file
def get_voxel_dimension_and_TR(in_file):
    import nibabel
    f = nibabel.load(in_file)
    return f.get_header()['pixdim'][1:4].tolist(), f.get_header()['pixdim'][4]

# Create the function Node
voxeldim = pe.Node(Function(input_names=['in_file'],
                            output_names=['voxel_dim', 'TR'],
                            function=get_voxel_dimension_and_TR),
                   name='voxeldim')

# To test this new node, feed the absolute path to the in_file as input
voxeldim.inputs.in_file = '~/nipype_tutorial/data/sub001/run001.nii.gz'

# Run the node and save the executed node under red
res = voxeldim.run()

# Look at the outputs of the executed node
res.outputs

# And this is the output you will see
```

```
Out[1]: TR = 2000.0
        voxel_dim = [3.0, 3.0, 4.0]
```

Note: For more information about the function `Function`, see this section (http://nipy.org/nipype/users/function_interface.html) on the official homepage.

Function Free Nodes

Sometimes you need a Node without a specific interface function. A Node that just distributes values. For example, when you need to feed the voxel dimension and the different subject names into your pipeline. Don't worry that you'll need a complex node to do this. You only need a Node that can receive the input `[3.0, 3.0, 4.0]` and `['sub001','sub002','sub003']` and distribute those inputs to the workflow.

Such a way of identity mapping input to output can be achieved with Nipype's own `IdentityInterface` function from the `utility` interface:

```python
# Import necessary modules
import nipype.pipeline.engine as pe      # import pypeline engine
import nipype.interfaces.utility as util # import the utility interface

# Create the function free node with specific in- and output fields
identitynode = pe.Node(util.IdentityInterface(fields=['subject_name',
                                                     'voxel_dimension']),
                       name='identitynode')

# Specify certain values of those fields
identitynode.inputs.voxel_dimension = [3.0, 3.0, 4.0]

# Or use iterables to distribute certain values
identitynode.iterables = ('subject_name', ['sub001','sub002','sub003'])
```

9.2.4 Specify Workflows & Connect Nodes

Workflows

Workflows are the scaffolds of a pipeline. They are, together with Nodes, the core element of any pipeline. The purpose of workflows is to guide the sequential execution of Nodes. This is done by connecting Nodes to the workflow and to each other in a certain way. The nice thing about workflows is, that they themselves can be connected to other workflows or can be used as a sub part of another, bigger worklfow. So how are they actually created?

Workflows are implemented almost the same as Nodes are. Except that you don't need to declare any interface or function:

```python
workflowname = Workflow(name='label')
```

This is all you have to do.

Establish Connections

But just creating workflows is not enough. You also have to tell it which nodes to connect with which other nodes and therefore specify the direction and order of execution.

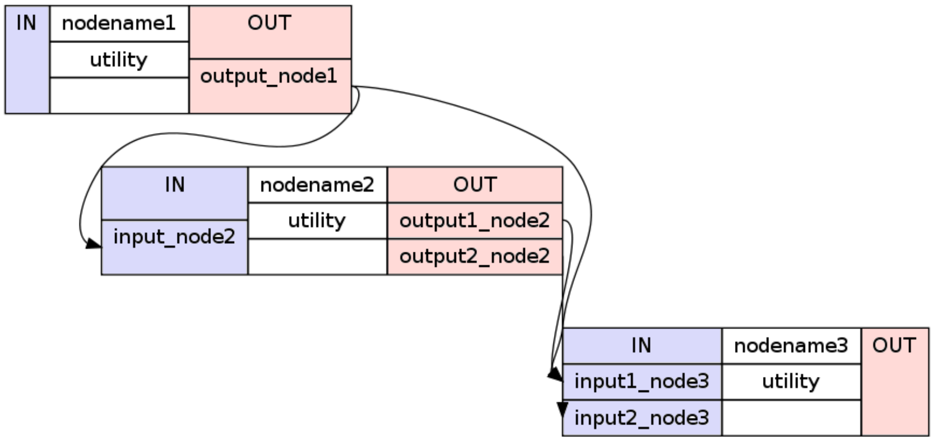
Connect Nodes to Nodes

There is a basic and an advanced way how to create connections between two nodes. The basic way allows only to connect two nodes at a time whereas the advanced way 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')
                                             ])
                      ])
```

It is important to point out that you do not only have to connect the nodes, but rather that you have to connect the output and input fields of each node to the output and input fields of another node.

If you visualize the advanced connection example as a detailed graph, which will be covered in the next section, it would look something as follows:



Connect Workflows to Workflows

Sometimes you also want to connect a workflow to another workflow. For example a preprocessing pipeline to a analysis pipeline. This, so that you can execute the whole pipeline as one. To do this, you can't just connect the nodes to each other. You have to additionally connect the workflows to themselves.

Let's assume that we have a node `realign` which is part of a preprocessing pipeline called `preprocess` and that we have a node called `modelspec` which is part of an analysis pipeline called `modelestimation`. To be able to connect those two pipelines at those particular points we need another workflow to serve as a connection scaffold:

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

As you see, the main difference to the connections between nodes is that you connect the pipelines first. Nonetheless, you still have to specify which nodes with which output or input fields have to be connected to each other.

Add stand-alone Nodes to Workflow

There is also the option to add nodes to a workflow without really connecting them to any other nodes or workflow. This can be done with the `add_nodes` function.

For example

```
#Add smooth and realign to the workflow
workflow.add_nodes([smooth, realign])
```

Modify Values between Connections

Sometimes you want to modify the output of one node before sending it on to the next node. This can be done in two ways. Either use an individual node as described above (http://miykael.github.io/nipype-beginner-s-guide/firstSteps.html#individual-nodes), or plant a function directly between the output and input of two nodes. To do the second approach, do as follows:

First, define your function that modifies the data as you want and returns the new output:

```
# Define your function that does something special
def myfunction(output_from_node1):
    input_for_node2 = output_from_node1 * 2
    return input_for_node2
```

Second, insert this function between the connection of the two nodes of interest:

```
# Insert function between the connection of the two nodes
workflow.connect([(nodename1, nodename2,[(('output_from_node1', myfunction),
                                           'input_for_node2')]),
                  ])
```

This will take the output of `output_from_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 `input_from_node2`.

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

```
# Define your function that does something special
def myfunction(output_from_node1, additional_input):
    input_for_node2 = output_from_node1 + additional_input
    return input_for_node2

# Insert function with additional input between the connection of the two nodes
workflowname.connect([(nodename1,nodename2,[(('out_file_node1', myfunction,
                                               additional_input),
                                              'input_for_node2')]),
                      ])
```

Clone Existing Pipelines

Sometimes you want to reuse a pipeline you've already created with some different parameters and node connections. Instead of just copying and changing the whole script, just use the `clone` command.

For example, if you've already created an analysis pipeline that analysis the data on the volume and now would love to reuse this pipeline to do the analysis of the surface, just do as follows:

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

This is all you have to do to have the same connections and parameters in `surfanalysis` as you have in `volanalysis`. If you wouldn't clone the pipeline and keep continuing the same pipeline, Nipype would assume that it still is the same execution flow and just rewrite all the output from the `volanalysis` pipeline.

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

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

9.2.5 Input & Output Stream

This is probably one of the more important and difficult sections of a workflow script, as most of the errors and issues you can encounter with your pipeline are mostly based on some kind of error in the specification of the workflow input or output stream. So make sure that this section is correct.

Before you can tell your computer where it can find your data, you yourself have to understand where and in which format your data is stored at. If you use the tutorial dataset, then your folder structure look as follows:

```
nipype_tutorial
|-- data
|    |-- sub001
|    |    |-- ...
|    |    |-- run001.nii.gz
|    |    |-- run002.nii.gz
|    |    |-- struct.nii.gz
|    |-- sub002
|    |-- sub003
|-- freesurfer
     |-- sub001
     |-- sub002
     |-- sub003
```

So this means that your scans are stored in a zipped NIfTI format (i.e. `nii.gz`) and that you can find them as follows: `~/nipype_tutorial/data/subjectname/scanimage.nii.gz`

Input Stream

Now there are two different functions that you can use to specify the folder structure of the input stream. One of them is called `SelectFiles` and the other one is called `DataGrabber`. Both are string based and easy to use once understood. Nonetheless, I would recommend to use `SelectFiles`, as it is much more straight forward to use:

```
import nipype.interfaces.io as nio

# SelectFiles
templates = {'anat': 'data/{subject_id}/struct.nii.gz',
             'func': 'data/{subject_id}/run*.nii.gz'}
selectfiles = Node(nio.SelectFiles(templates), name="selectfiles")

# DataGrabber
datasource = Node(nio.DataGrabber(infields=['subject_id'],
                                  outfields=['anat', 'func'],
                                  template = 'data/%s/%s.nii'),
                  name = 'datasource')
```

```

info = dict(anat=[['subject_id', 'struct']],
            func=[['subject_id', ['run001','run002']]])

datasource.inputs.template_args = info
datasource.inputs.sort_filelist = True
```

Note: Go to the official homepage to read more about DataGrabber (http://nipy.org/nipype/users/grabbing_and_sinking.html#datagrabber) and SelectFiles (http://nipy.org/nipype/users/select_files.html).

Output Stream

In contrast to this, the definition of the output stream is rather simple. You only have to create a `DataSink`. A `DataSink` is a node that specifies in which output folder all the relevant results should be stored at.

```
# Datasink
datasink = Node(nio.DataSink(), name="datasink")
datasink.inputs.base_directory = '~/nipype_tutorial'
datasink.inputs.container = 'datasink_folder'
```

To store an output of a certain node in this DataSink just connect the node to the DataSink. The output data will be saved in the just specified container `datasink_folder`. Nipype will then save this output in a folder under this container, depending on the name of the DataSink input field that you specify during the creation of connections.

As an example, let's assume that we want to use the output of SPM's motion correction node, here called `realign`.

```
# Saves the realigned files into a subfolder called 'motion'
workflow.connect(realign, datasink, [('realigned_files', 'motion')])

# Saves the realignment_parameters also into the subfolder called 'motion'
workflow.connect(realign, datasink, [('realignment_parameters', 'motion.@par')])

# Saves the realignment parameters in a subfolder 'par', under the folder 'motion'
workflow.connect(realign, datasink, [('realignment_parameters', 'motion.par')])
```

The output folder and files of the datasink node often have long and detailed names, such as `'_subject_id_sub002/con_0001_warped_out.nii'`. This is because many of the nodes used add their own pre- or postfix to a file or folder. You can use datasink's substitutions function to change or delete unwanted strings:

```
# Use the following DataSink output substitutions
substitutions = [('_subject_id_', ''),
                 ('warped_out', 'final')]
datasink.inputs.substitutions = substitutions
```

This substition will change `'_subject_id_sub002/con_0001_warped_out.nii'` into `'sub002/con_0001_final.nii'`.

The DataSink is really useful to keep control over your storage capacity. If you store all the important outputs of your workflow in this folder, you can delete the workflow working directory 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 (http://nipy.org/nipype/users/config_file.html).

Note: Go to the official homepage to read more about DataSink (http://nipy.org/nipype/users/grabbing_and_sinking.html#datasink).

Important: After you've created the input and output node it is very important to connect them to the rest of your workflow. Otherwise your pipeline would have no real input or output stream. You can see how to do this in the example below.

9.2.6 Run Workflow

After all modules are imported, important variables are specified, nodes are created and connected to workflows, you are able to run your pipeline. This can be done by calling the `run()` method of the workflow.

As already described in the introduction section (http://miykael.github.io/nipype-beginner-s-guide/nipype.html#execution-plugins), workflows can be run with many different plugins. Those plugins allow you to run your workflow in either normal linear (i.e. sequential) or in parallel ways. Depending on your system, parallel execution is either done on your local machine or on some computation cluster.

Here are just a few example how you can run your workflow:

```
# Execute your workflow in sequential way
workflow.run()

# Execute your workflow in parallel.
#   Use 4 cores on your local machine
workflow.run('MultiProc', plugin_args={'n_procs': 4})

#   Use a cluster environment to run your workflow
workflow.run('SGE', plugin_args={'qsub_args': '-q many'})
```

The computation time of your workflow depends on many different factors, such as which nodes you use, with which parameters, on how many subjects, if you use parallel execution and the power of your system. Therefore, no real prediction about the execution time can be made.

But the nice thing about Nipype is that it will always check if a node has already been run and if the input parameters have changed or not. Only nodes that have different input parameters will be rerun. Nipype's hashing mechanism ensures that none of the nodes are executed during a new run if the inputs remain the same. This keeps the computation time to its minimum.

Note: More about Plugins and how you can run your pipeline in a distributed system can be found on the official homepage under Using Nipype Plugins (http://nipy.org/nipype/users/plugins.html).

9.3 Example Script

Let's try to summarize what we've learned by building a short preprocessing pipeline. The following script assumes that you're using the tutorial dataset with the three subjects `sub001`, `sub002` and `sub003`, each having two functional scans `run001.nii.gz` and `run002.nii.gz`.

9.3.1 Import modules

First, import all necessary modules. Which modules you have to import becomes clear while you're adding specific nodes.

```python
from os.path import join as opj
from nipype.interfaces.spm import SliceTiming, Realign, Smooth
from nipype.interfaces.utility import IdentityInterface
from nipype.interfaces.io import SelectFiles, DataSink
from nipype.algorithms.rapidart import ArtifactDetect
from nipype.algorithms.misc import Gunzip
from nipype.pipeline.engine import Workflow, Node
```

9.3.2 Specify variables

Specify all variables that you want to use later in the script. This makes the modification between experiments easy.

```python
experiment_dir = '~/nipype_tutorial'             # location of experiment folder
data_dir = opj(experiment_dir, 'data')  # location of data folder
fs_folder = opj(experiment_dir, 'freesurfer')  # location of freesurfer folder

subject_list = ['sub001', 'sub002', 'sub003']    # list of subject identifiers
session_list = ['run001', 'run002']              # list of session identifiers

output_dir = 'output_firstSteps'          # name of output folder
working_dir = 'workingdir_firstSteps'     # name of working directory

number_of_slices = 40                     # number of slices in volume
TR = 2.0                                  # time repetition of volume
smoothing_size = 8                        # size of FWHM in mm
```

9.3.3 Specify Nodes

Let's now create all the nodes we need for this preprocessing workflow:

- **Gunzip**: This node is needed to convert the NIfTI files from the zipped version `.nii.gz` to the unzipped version `.nii`. This step has to be done because SPM's SliceTiming can not handle zipped files.
- **SliceTiming**: This node executes SPM's SliceTiming on each functional scan.
- **Realign**: This node executes SPM's Realign on each slice time corrected functional scan.
- **ArtifactDetect**: This node executes ART (http://www.nitrc.org/projects/artifact_detect/)'s artifact detection on the functional scans.
- **Smooth**: This node executes SPM's Smooth on each realigned functional scan.

```python
# Gunzip - unzip functional
gunzip = Node(Gunzip(), name="gunzip")

# Slicetiming - correct for slice wise acquisition
interleaved_order = range(1,number_of_slices+1,2) + range(2,number_of_slices+1,2)
sliceTiming = Node(SliceTiming(num_slices=number_of_slices,
                               time_repetition=TR,
                               time_acquisition=TR-TR/number_of_slices,
                               slice_order=interleaved_order,
                               ref_slice=2),
11                     name="sliceTiming")
12
13 # Realign - correct for motion
14 realign = Node(Realign(register_to_mean=True),
15                name="realign")
16
17 # Artifact Detection - determine which of the images in the functional series
18 #   are outliers. This is based on deviation in intensity or movement.
19 art = Node(ArtifactDetect(norm_threshold=1,
20                           zintensity_threshold=3,
21                           mask_type='spm_global',
22                           parameter_source='SPM'),
23            name="art")
24
25 # Smooth - to smooth the images with a given kernel
26 smooth = Node(Smooth(fwhm=smoothing_size),
27               name="smooth")
```

Note: **Line 5** specifies the slice wise scan acquisition. In our case this was interleaved ascending. Use the following code if you just have ascending (`range(1,number_of_slices+1)`) or descending (`range(number_of_slices,0,-1)`).

9.3.4 Specify Workflows & Connect Nodes

After we've created all the nodes we can create our preprocessing workflow and connect the nodes to this workflow.

```
1  # Create a preprocessing workflow
2  preproc = Workflow(name='preproc')
3  preproc.base_dir = opj(experiment_dir, working_dir)
4
5  # Connect all components of the preprocessing workflow
6  preproc.connect([(gunzip, sliceTiming, [('out_file', 'in_files')]),
7                   (sliceTiming, realign, [('timecorrected_files', 'in_files')]),
8                   (realign, art, [('realigned_files', 'realigned_files'),
9                                   ('mean_image', 'mask_file'),
10                                  ('realignment_parameters',
11                                   'realignment_parameters')]),
12                  (realign, smooth, [('realigned_files', 'in_files')]),
13                  ])
```

Note: **Line 3** is needed to tell the workflow in which folder it should be run. You don't have to do this for any subworkflows that you're using. But the "main" workflow, the one which we execute with the `.run()` command, should always have a `base_dir` specified.

9.3.5 Input & Output Stream

Before we can run our preprocessing workflow, we first have to specify the input and output stream. To do this, we first have to create the distributor node `Infosource`, the input node `SelectFiles` and the output node `DataSink`. The purpose of `Infosource` is to tell `SelectFiles` over which elements of its input stream it should iterate over.

To finish it all up, those three nodes now have to be connected to the rest of the pipeline.

```python
# Infosource - a function free node to iterate over the list of subject names
infosource = Node(IdentityInterface(fields=['subject_id',
                                            'session_id']),
                  name="infosource")
infosource.iterables = [('subject_id', subject_list),
                        ('session_id', session_list)]

# SelectFiles
templates = {'func': 'data/{subject_id}/{session_id}.nii.gz'}
selectfiles = Node(SelectFiles(templates,
                               base_directory=experiment_dir),
                   name="selectfiles")

# Datasink
datasink = Node(DataSink(base_directory=experiment_dir,
                         container=output_dir),
                name="datasink")

# Use the following DataSink output substitutions
substitutions = [('_subject_id', ''),
                 ('_session_id_', '')]
datasink.inputs.substitutions = substitutions

# Connect SelectFiles and DataSink to the workflow
preproc.connect([(infosource, selectfiles, [('subject_id', 'subject_id'),
                                            ('session_id', 'session_id')]),
                 (selectfiles, gunzip, [('func', 'in_file')]),
                 (realign, datasink, [('mean_image', 'realign.@mean'),
                                      ('realignment_parameters',
                                       'realign.@parameters'),
                                      ]),
                 (smooth, datasink, [('smoothed_files', 'smooth')]),
                 (art, datasink, [('outlier_files', 'art.@outliers'),
                                  ('plot_files', 'art.@plot'),
                                  ]),
                 ])
```

9.3.6 Run Workflow

Running the pipeline is a rather simple thing. Just use the `.run()` command with the plugin you want. In our case we want to preprocess the 6 functional scans on 6 cores at once.

```python
preproc.write_graph(graph2use='flat')
preproc.run('MultiProc', plugin_args={'n_procs': 6})
```

As you see, we've executed the function `write_graph()` before we've run the pipeline. `write_graph()` is not needed to run the pipeline, but allows you to visualize the execution flow of your pipeline, before you actually execute the pipeline. More about the visualization of workflows can be found in the next chapter, How To Visualize A Pipeline (http://miykael.github.io/nipype-beginner-s-guide/visualizePipeline.html).

Hint: You can download the code for this preprocessing pipeline as a script here: tutorial_3_first_steps.py (https://github.com/miykael/nipype-beginner-s-guide/blob/master/scripts/tutorial_3_first_steps.py)

9.4 Resulting Folder Structure

After we've executed the preprocessing pipeline we have two new folders under `~/nipype_tutorial`. The working directory `workingdir_firstSteps` which contains all files created during the execution of the workflow, and the output folder `output_firstSteps` which contains all the files that we sent to the DataSink. Let's take a closer look at those two folders.

9.4.1 Working Directory

The working directory contains many temporary files that might be not so important for your further analysis. That's why I highly recommend to save all the important outputs of your workflow in a DataSink folder. So that everything important is at one place.

The following folder structure represents the working directory of the above preprocessing workflow:

```
workingdir_firstSteps
|-- preproc
    |-- _session_id_run001_subject_id_sub001
    |    |-- art
    |    |-- datasink
    |    |-- gunzip
    |    |-- realign
    |    |-- selectfiles
    |    |-- sliceTiming
    |    |-- smooth
    |-- _session_id_run001_subject_id_sub002
    |-- _session_id_run001_subject_id_sub003
    |-- _session_id_run002_subject_id_sub001
    |-- _session_id_run002_subject_id_sub002
    |-- _session_id_run002_subject_id_sub003
```

Even though the working directory is most often only temporary, it contains many relevant files to be found and explore. Following are some of the highlights:

- **Visualization**: The main folder of the workflow contains the visualized graph files (if created with `write_graph()` and an interactive execution fiew (`index.html`).
- **Reports**: Each node contains a subfolder called `_report` that contains a file called `report.rst`. This file contains all relevant node information. E.g. What's the name of the node and what is its hierarchical place in the pipeline structure? What are the actual input and executed output parameters? How long did it take to execute the node and what were the values of the environment variables during the execution?

9.4.2 Output Folder

The output folder contains exactly the files that we sent to the DataSink. Each node contains its own folder and in each of those folder a subfolder for each subject is created.

```
output_firstSteps
|-- art
|    |-- run001_sub001
|    |    |-- art.rarun001_outliers.txt
|    |    |-- plot.rarun001.png
|    |-- run001_sub002
|    |-- run001_sub003
|    |-- run002_sub001
|   |-- run002_sub002
|   |-- run002_sub003
|-- realign
|   |-- run001_sub001
|   |   |-- meanarun001.nii
|   |   |-- rp_arun001.txt
|   |-- run001_sub002
|   |-- run001_sub003
|   |-- run002_sub001
|   |-- run002_sub002
|   |-- run002_sub003
|-- smooth
    |-- run001_sub001
    |   |-- srarun001.nii
    |-- run001_sub002
    |-- run001_sub003
    |-- run002_sub001
    |-- run002_sub002
    |-- run002_sub003
```

The goal of this output folder is to store all important outputs in this folder. This allows you to delete the working directory and get rid of its many unnecessary temporary files.

9.5 Common Issues, Problems and Crashes

As so often in life, there is always something that doesn't go as planed. And this is the same for Nipype. There are many reasons why a pipeline can cause problems or even crash. But there's always a way to figure out what went wrong and what needs to be fixed.

9.5.1 Best Case Scenario - Everything Works

Before we take a look at how to find errors, let's take a look at a correct working pipeline. The following is the abbreviated terminal output of the preprocessing workflow from above. For readability reasons, lines containing the execution timestamps are not shown:

```
['check', 'execution', 'logging']
Running in parallel.
Submitting 3 jobs
Executing: selectfiles.a2 ID: 0
Executing: selectfiles.a1 ID: 1
Executing: selectfiles.a0 ID: 14
Executing node selectfiles.a2 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                        preproc/_subject_id_sub003/selectfiles
Executing node selectfiles.a1 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                        preproc/_subject_id_sub002/selectfiles
Executing node selectfiles.a0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                        preproc/_subject_id_sub001/selectfiles
[Job finished] jobname: selectfiles.a2 jobid: 0
[Job finished] jobname: selectfiles.a1 jobid: 1
[Job finished] jobname: selectfiles.a0 jobid: 14

[...]

Submitting 3 jobs
```

```
Executing: datasink.a1 ID: 7
Executing: datasink.a2 ID: 13
Executing: datasink.a0 ID: 20
Executing node datasink.a0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                    preproc/_subject_id_sub001/datasink
Executing node datasink.a1 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                    preproc/_subject_id_sub002/datasink
Executing node datasink.a2 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                    preproc/_subject_id_sub003/datasink
[Job finished] jobname: datasink.a1 jobid: 7
[Job finished] jobname: datasink.a2 jobid: 13
[Job finished] jobname: datasink.a0 jobid: 20
```

This output shows you the chronological execution of the pipeline, run in parallel mode. Each node first has to be transformed into a job and submitted to the execution cluster. The start of a node's execution is accompanied by the working directory of this node. The output `[Job finished]` then tells you when the execution of the node is done.

9.5.2 It Crashes, But Where is the Problem?

In the beginning when you're not used to reading Nipype's terminal output it can be tricky to find the actual error. But most of the time, Nipype tells you exactly what's wrong.

Let's assume for example, that you want to create a preprocessing pipeline as shown above but forget to provide the mandatory input `realigned_files` for the artifact detection node `art`. The running of such a workflow will lead to the following terminal output:

```
141018-14:01:51,671 workflow INFO:
     ['check', 'execution', 'logging']
141018-14:01:51,688 workflow INFO:
     Running serially.
141018-14:01:51,689 workflow INFO:
     Executing node selectfiles.b0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                            preproc/_subject_id_sub001/selectfiles
141018-14:01:51,697 workflow INFO:
     Executing node gunzip.b0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                       preproc/_subject_id_sub001/gunzip
141018-14:01:51,699 workflow INFO:
     Executing node _gunzip0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                      preproc/_subject_id_sub001/gunzip/mapflow/_
↪gunzip0
141018-14:01:52,53 workflow INFO:
     Executing node sliceTiming.b0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                            preproc/_subject_id_sub001/sliceTiming
141018-14:02:30,30 workflow INFO:
     Executing node realign.b0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                        preproc/_subject_id_sub001/realign
141018-14:03:29,374 workflow INFO:
     Executing node smooth.aI.a1.b0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                             preproc/_subject_id_sub001/smooth
141018-14:04:40,927 workflow INFO:
     Executing node art.b0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                    preproc/_subject_id_sub001/art
141018-14:04:40,929 workflow ERROR:
     ['Node art.b0 failed to run on host mnotter.']
141018-14:04:40,930 workflow INFO:
     Saving crash info to ~/nipype_tutorial/crash-20141018-140440-mnotter-art.b0.pklz
141018-14:04:40,930 workflow INFO:
```

```
31      Traceback (most recent call last):
32    File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/plugins/linear.py",
33          line 38, in run node.run(updatehash=updatehash)
34    File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/engine.py",
35          line 1424, in run self._run_interface()
36    File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/engine.py",
37          line 1534, in _run_interface self._result = self._run_command(execute)
38    File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/engine.py",
39          line 1660, in _run_command result = self._interface.run()
40    File "~/anaconda/lib/python2.7/site-packages/nipype/interfaces/base.py",
41          line 965, in run self._check_mandatory_inputs()
42    File "~/anaconda/lib/python2.7/site-packages/nipype/interfaces/base.py",
43          line 903, in _check_mandatory_inputs raise ValueError(msg)
44  ValueError: ArtifactDetect requires a value for input 'realigned_files'.
45              For a list of required inputs, see ArtifactDetect.help()
46
47  141018-14:05:18,144 workflow INFO:
48       ***********************************
49  141018-14:05:18,144 workflow ERROR:
50       could not run node: preproc.art.b0
51  141018-14:05:18,144 workflow INFO:
52       crashfile: ~/nipype_tutorial/crash-20141018-140440-mnotter-art.b0.pklz
53  141018-14:05:18,144 workflow INFO:
54       ***********************************
```

Now, what happened? **Line 26** indicates you that there is an Error and **line 29** tells you where the crash report to this error was saved at. The last part of this crash file (i.e. `art.b0.pklz`) tells you that the error happened in the `art` node. **Line 31-43** show the exact error stack of the current crash. Those multiple lines starting with `File` are also always a good indicator to find the error in the terminal output.

Now the important output is shown in **line 44**. Here it actually tells you what is wrong. `ArtifactDetect requires a value for input 'realigned_files'`. Correct this issue and the workflow should execute cleanly.

Always at the end of the output is a section that summarizes the whole crash. In this case this is **line 48-54**. Here you can see again which nodes lead to the crash and where the crash file to the error is stored at.

9.5.3 Read the Crash File

But sometimes, just knowing where and because of what the crash happened is not enough. You also need to know what the actual values of the crashed nodes were, to see if perhaps some input values were not transmitted correctly.

This can be done with the shell command `nipype_display_crash`. To read for example the above mentioned `art`-crash file, we have to open a new terminal and run the following command:

```
nipype_display_crash ~/nipype_tutorial/crash-20141018-140857-mnotter-art.b0.pklz
```

This will lead to the following output:

```
File: crash-20141018-140857-mnotter-art.b0.pklz
Node: preproc.art.b0
Working directory: ~/nipype_tutorial/workingdir_firstSteps/
                     preproc/_subject_id_sub001/art

Node inputs:

bound_by_brainmask = False
```

```
global_threshold = 8.0
ignore_exception = False
intersect_mask = <undefined>
mask_file = <undefined>
mask_threshold = <undefined>
mask_type = spm_global
norm_threshold = 0.5
parameter_source = SPM
plot_type = png
realigned_files = <undefined>
realignment_parameters = ['~/nipype_tutorial/workingdir_firstSteps/preproc/
                              _subject_id_sub001/realign/rp_arun001.txt']
rotation_threshold = <undefined>
save_plot = True
translation_threshold = <undefined>
use_differences = [True, False]
use_norm = True
zintensity_threshold = 3.0

Traceback (most recent call last):
  File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/plugins/linear.py",
       line 38, in run node.run(updatehash=updatehash)
  File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/engine.py",
       line 1424, in run self._run_interface()
  File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/engine.py",
       line 1534, in _run_interface self._result = self._run_command(execute)
  File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/engine.py",
       line 1660, in _run_command result = self._interface.run()
  File "~/anaconda/lib/python2.7/site-packages/nipype/interfaces/base.py",
       line 965, in run self._check_mandatory_inputs()
  File "~/anaconda/lib/python2.7/site-packages/nipype/interfaces/base.py",
       line 903, in _check_mandatory_inputs raise ValueError(msg)
ValueError: ArtifactDetect requires a value for input 'realigned_files'.
            For a list of required inputs, see ArtifactDetect.help()
```

From this output you can see in the lower half the same error stack of the crash and the exact description of what is wrong, as we've seen in the terminal output. But in the first half you also have additional information of the nodes input, which might help to solve some problems.

Note: Note that the information about the exact input values of a node can also be obtained from the `report.rst` file, stored in the nodes subfolder under the working directory. More about this later under Working Directory (http://miykael.github.io/nipype-beginner-s-guide/firstSteps.html#working-directory).

9.5.4 Interface Issues

Sometimes the most basic errors can occur because Nipype doesn't know where the correct files are. Two very common issues are for example that FreeSurfer can't find the subject folder or that MATLAB doesn't find SPM.

Before you do anything else, please make sure again that you've installed FreeSurfer and SPM12 as described in the installation section, How to install FreeSurfer (http://miykael.github.io/nipype-beginner-s-guide/installation.html#freesurfer) and How to install SPM (http://miykael.github.io/nipype-beginner-s-guide/installation.html#spm12).

But don't worry if the problem still exists. There are two nice ways how you can tell Nipype where FreeSurfer subject folders are stored at and where MATLAB can find SPM12. Just add the following code to the beginning of your script:

```python
# Import FreeSurfer and specify the path to the current subject directory
import nipype.interfaces.freesurfer as fs
fs.FSCommand.set_default_subjects_dir('~/nipype_tutorial/freesurfer')

# Import MATLAB command and specify the path to SPM12
from nipype.interfaces.matlab import MatlabCommand
MatlabCommand.set_default_paths('/usr/local/MATLAB/R2014a/toolbox/spm12')
```

9.5.5 Be Aware of Your Data

Sometimes the biggest issue with your code is that you try to force things that can't work.

One common example is that you try to feed in two values (e.g. `run001` and `run002`) into an input field that only expects one value. An example output of such an Error would contain something like this:

```
TraitError: The 'in_file' trait of a GunzipInputSpec instance must be an existing file
            name, but a value of ['~/nipype_tutorial/data/sub002/run001.nii.gz',
            '~/nipype_tutorial/data/sub002/run002.nii.gz'] <type 'list'> was␣
↪specified.
```

This error tells you that it expects an `'in_file'` (i.e. singular) and that the file `['run001', 'run002']` doesn't exist. What makes sense, because a *list* isn't a file. Such an error can often times be resolved by using a MapNode. Otherwise, find another way to reduce the number of input values per field.

Another common mistake is the fact that your data is given as input to a node that can't handle the format of this input. This error is most often encountered when your data type is zipped and the following node can't unzip the file by itself. We included a `Gunzip` node exaclty for this reason in the example pipeline above.

So what will the terminal output look like if we try to feed a zipped `run001.nii.gz` to a node that executes SPM's SliceTiming?

```
141018-13:25:22,227 workflow INFO:
     Executing node selectfiles.b0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                             preproc/_subject_id_sub001/selectfiles
141018-13:25:22,235 workflow INFO:
     Executing node sliceTiming.b0 in dir: ~/nipype_tutorial/workingdir_firstSteps/
                                             preproc/_subject_id_sub001/sliceTiming
141018-13:25:39,673 workflow ERROR:
     ['Node sliceTiming.b0 failed to run on host mnotter.']
141018-13:25:39,673 workflow INFO:
     Saving crash info to ~/nipype_tutorial/
                            crash-20141018-132539-mnotter-sliceTiming.b0.pklz

[...]

               < M A T L A B (R) >
     Copyright 1984-2014 The MathWorks, Inc.
       R2014a (8.3.0.532) 64-bit (glnxa64)
                February 11, 2014

[...]

Warning: Run spm_jobman('initcfg'); beforehand
> In spm_jobman at 106
  In pyscript_slicetiming at 362
Item 'Session', field 'val': Number of matching files (0) less than required (2).
Item 'Session', field 'val': Number of matching files (0) less than required (2).
```

```
27
28 Standard error:
29 MATLAB code threw an exception:
30 No executable modules, but still unresolved dependencies or incomplete module inputs.
31 File:/usr/local/MATLAB/R2014a/toolbox/spm12/spm_jobman.m
32 Name:/usr/local/MATLAB/R2014a/toolbox/spm12/spm_jobman.m
33 Line:47
34 File:~/nipype_tutorial/workingdir_firstSteps/preproc/
35        _subject_id_sub001/sliceTiming/pyscript_slicetiming.m
36 Name:fill_run_job
37 Line:115
38 File:pm_jobman
39 Name:pyscript_slicetiming
40 Line:459
41 File:ç
42 Name:U
43 Line:
44 Return code: 0
45 Interface MatlabCommand failed to run.
46 Interface SliceTiming failed to run.
47
48 141018-13:25:39,681 workflow INFO:
49      ***********************************
50 141018-13:25:39,682 workflow ERROR:
51      could not run node: preproc.sliceTiming
52 141018-13:25:39,682 workflow INFO:
53      crashfile: ~/nipype_tutorial/crash-20141018-132539-mnotter-sliceTiming.pklz
54 141018-13:25:39,682 workflow INFO:
55      ***********************************
```

This error message doesn't tell you directly what is wrong. Unfortunately, Nipype can't go deep enough into SPM's code and figure out what is wrong, because SPM itself doesn't tell us what's wrong. But **line 25-26** tells us that SPM has some issues with finding the appropriate files.

Hint: For more information about Errors go to the Support section (http://miykael.github.io/nipype-beginner-s-guide/index.html#support) of this beginner's guide.

CHAPTER

TEN

HOW TO VISUALIZE A PIPELINE

The option to visualize your workflow is a great feature of Nipype. It allows you to see your analysis in one piece, control the connections between the nodes and check which input and output fields are connected.

10.1 What kind of graph do you need?

You can visualize your pipeline as soon as you have a workflow that contains any nodes and connections between them. To create a graph, simply use the function `write_graph()` on your workflow:

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

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

- `orig` shows only the main workflows and omits any subworkflows
- `flat` shows all workflows, including any subworkflows
- `exec` shows all workflows, including subworkflows and expands iterables into subgraphs
- `hierarchical` shows all workflows, including subworkflows and also shows the hierarchical structure
- `colored` gives you the same output as `hierarchical` but color codes the different levels and the connections within those levels according to their hierarchical depth.

All types, except hierarchical and colored, create two graph files. The difference of those two files is in the level of detail they show. There is a **simple overview graph** called `graph.dot` which gives you the basic connections between nodes and a **more detailed overview graph** called `graph_detailed.dot` which additionally gives you the output and input fields of each node and the connections between them. The **hierarchical** and **colored graph** on the other side create only a simple overview graph. I mostly use **colored** graphs, as it gives you a fast and clear picture of your workflow structure.

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

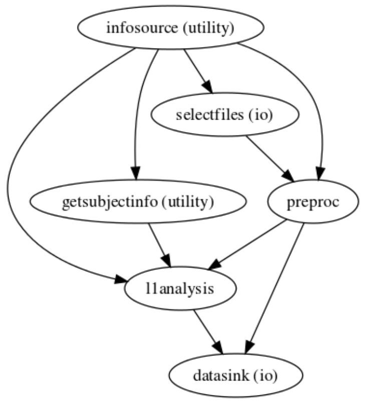
If graphviz is installed the dot files will automatically be converted into png-files. If not, take and load the dot files in any graphviz visualizer of your choice.

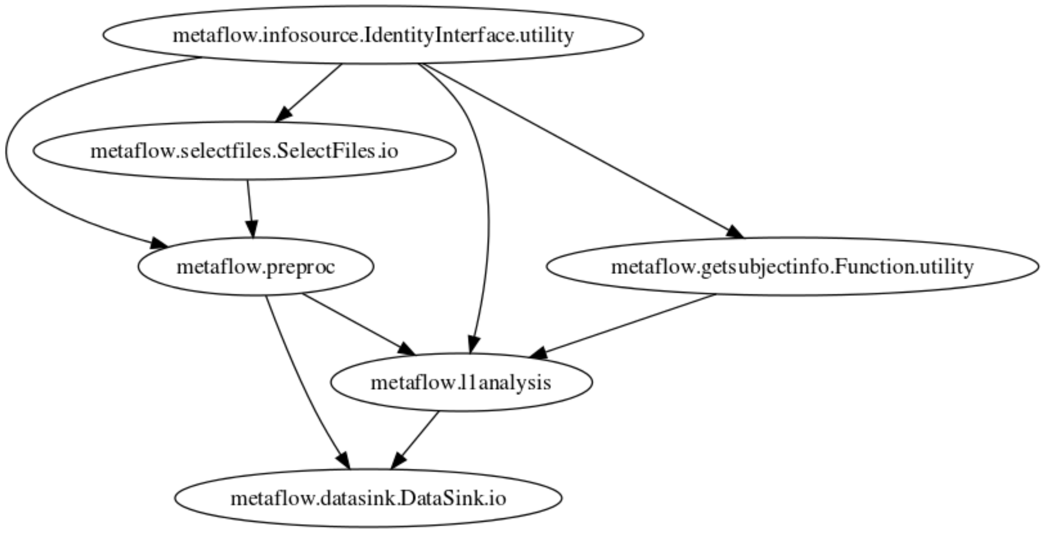
10.2 Tweak your visualization

There are two additional parameters `format` and `simple_form` that you can use to change your output graph. `format` can be used to change the output format of the image file to either `png` or `svg`. `simple_form` determines if the node name shown in the visualization is either of the form `nodename (package)` when set to `True` or `nodename.Class.package` when set to `False`.

```
workflow.write_graph(graph2use='colored', format='svg', simple_form=True)
```

To illustrate, on the left you can see a simple graph of the visualization type **orig** when `simple_form` is set to `True` and on the right if it is set to `False`.



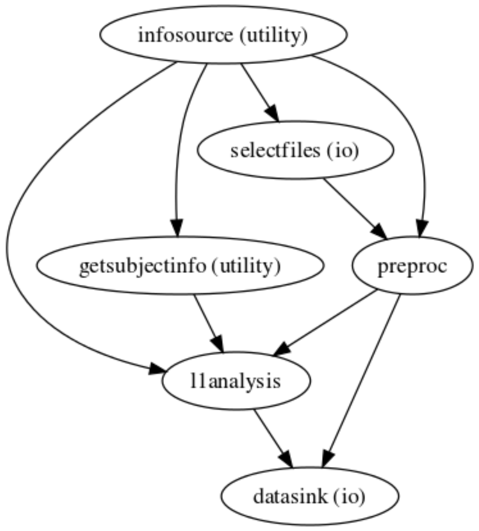

metaflow.infosource.IdentityInterface.utility
metaflow.selectfiles.SelectFiles.io
metaflow.preproc
metaflow.getsubjectinfo.Function.utility
metaflow.l1analysis
metaflow.datasink.DataSink.io

10.3 Examples of each visualization type

The graphs shown below are visualizations of the **first level analysis pipeline** or `metaflow` from the section: How To Build A First Level Pipeline (http://miykael.github.io/nipype-beginner-s-guide/firstLevel.html)

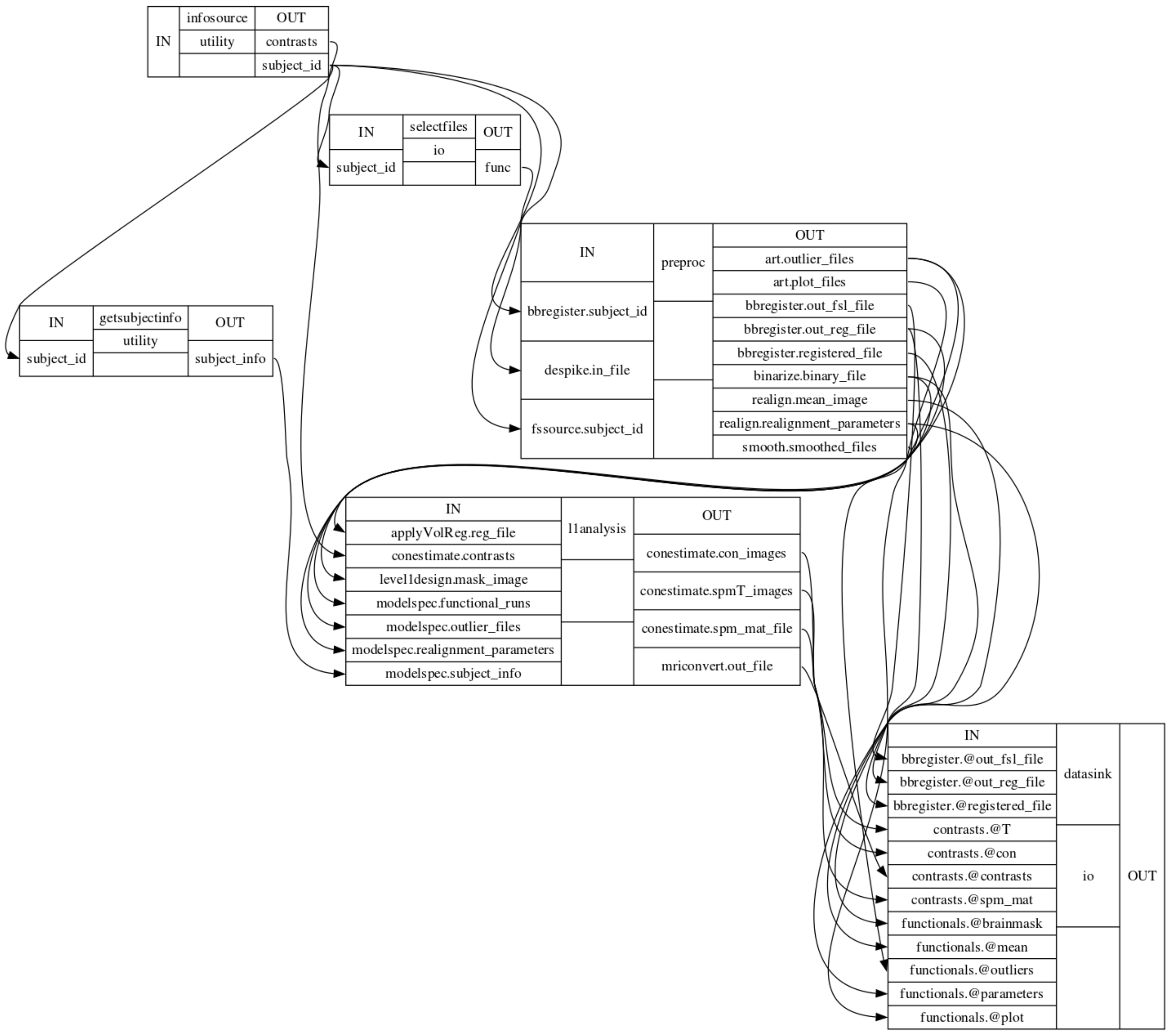
10.3.1 `orig` - simple graph

The simple graph of the visualization type `orig` shows only the top layer, i.e. hierarchical highest workflows and nodes, of your workflow. In this case this is the `metaflow`. Subworkflows such as `preproc` and `l1analysis1` are represented by a single node.



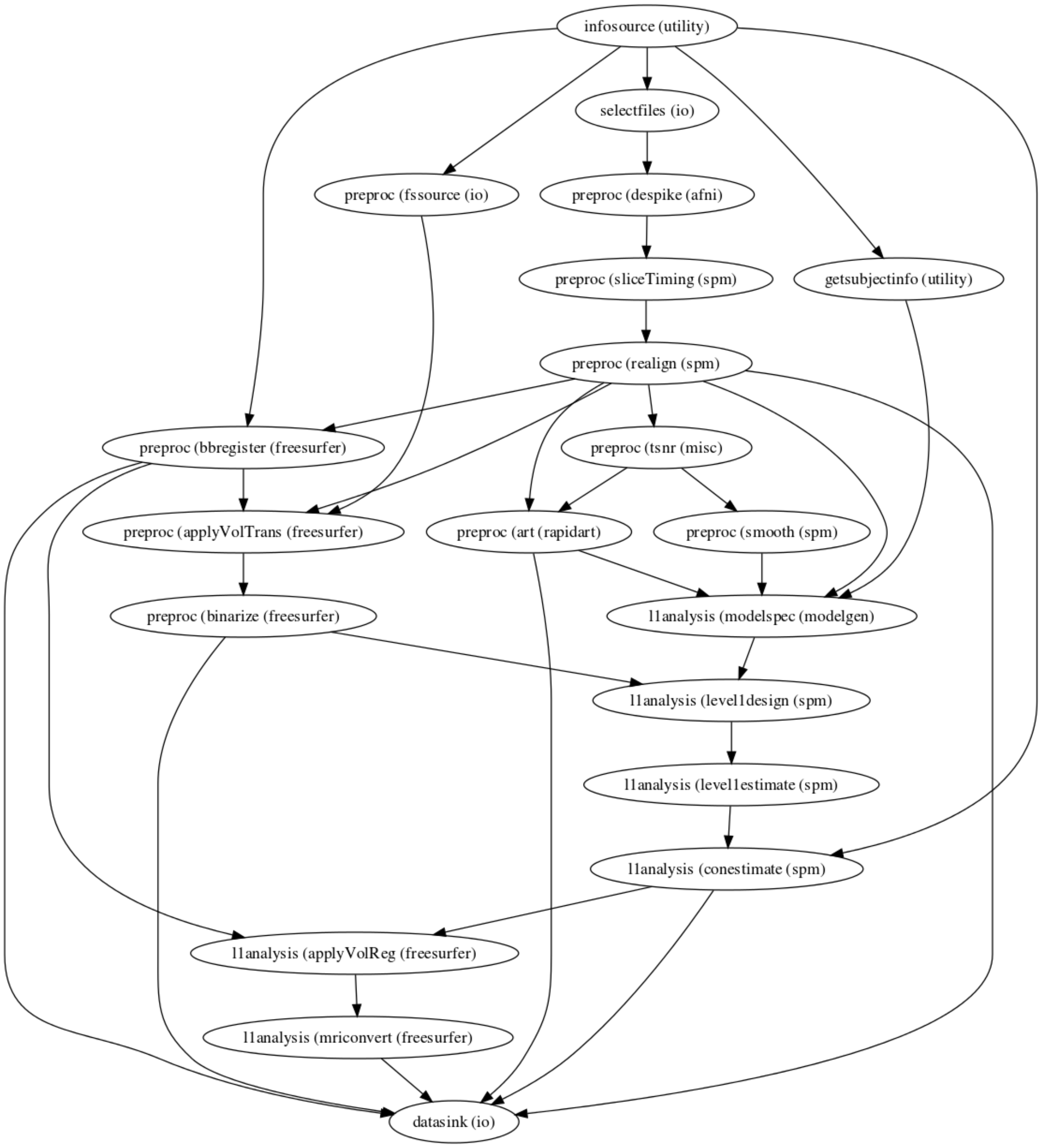
10.3.2 `orig` - detailed graph

The detailed graph of the visualization type `orig` shows the `metaflow` to the same depth as the simple version above, but with a bit more information about input and output fields. Now you can see which output of which node is connected to which input of the following node.



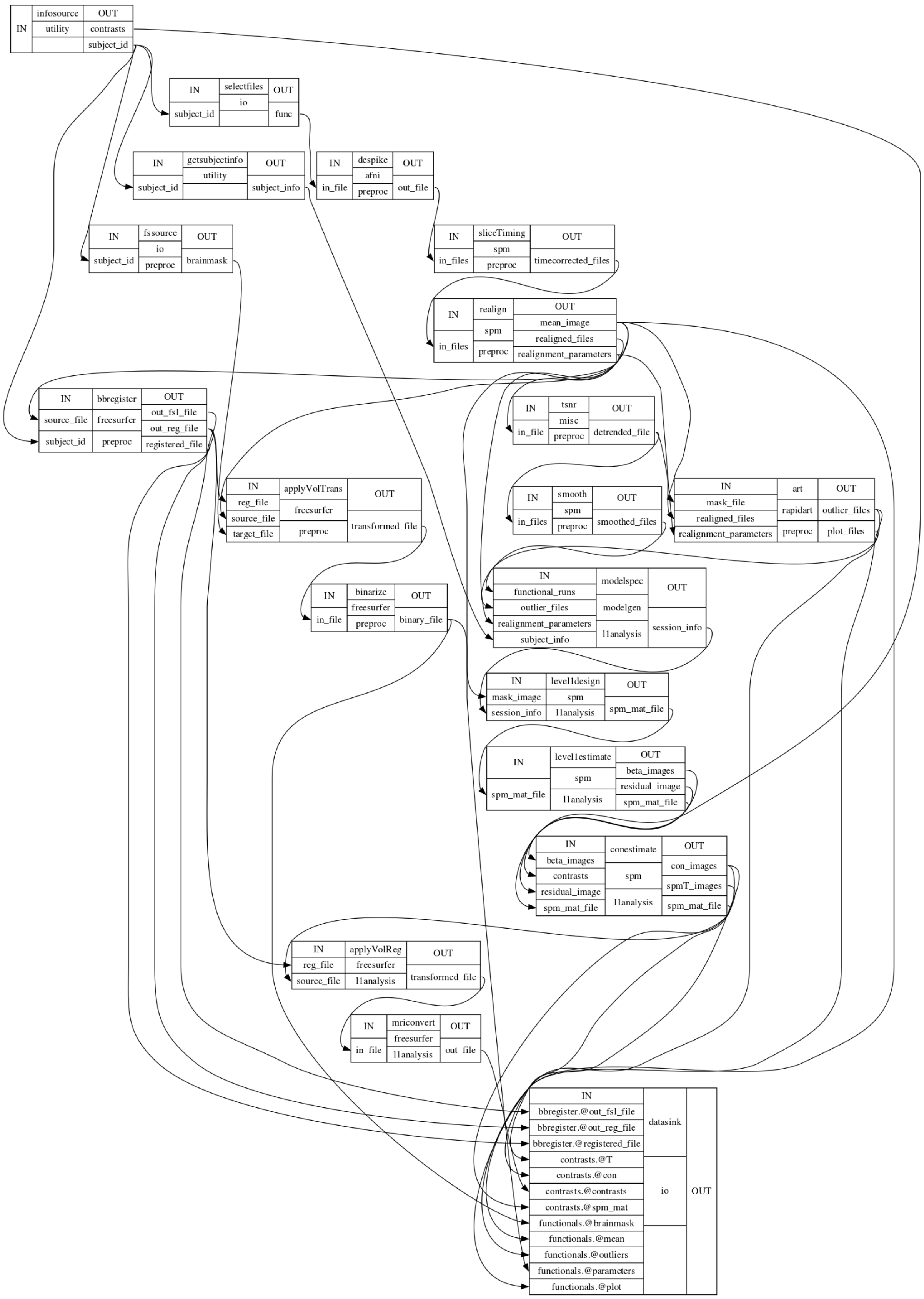
10.3.3 `flat` - simple graph

The simple graph of the visualization type `flat` shows all nodes of a workflow. As you can see, subworkflows such as `preproc` and `l1analysis1` are now expanded and represented by all their containing nodes.


infosource (utility)
selectfiles (io)
preproc (fssource (io)
preproc (despike (afni)
preproc (sliceTiming (spm)
getsubjectinfo (utility)
preproc (realign (spm)
preproc (bbregister (freesurfer)
preproc (tsnr (misc)
preproc (applyVolTrans (freesurfer)
preproc (art (rapidart)
preproc (smooth (spm)
preproc (binarize (freesurfer)
l1analysis (modelspec (modelgen)
l1analysis (level1design (spm)
l1analysis (level1estimate (spm)
l1analysis (conestimate (spm)
l1analysis (applyVolReg (freesurfer)
l1analysis (mriconvert (freesurfer)
datasink (io)

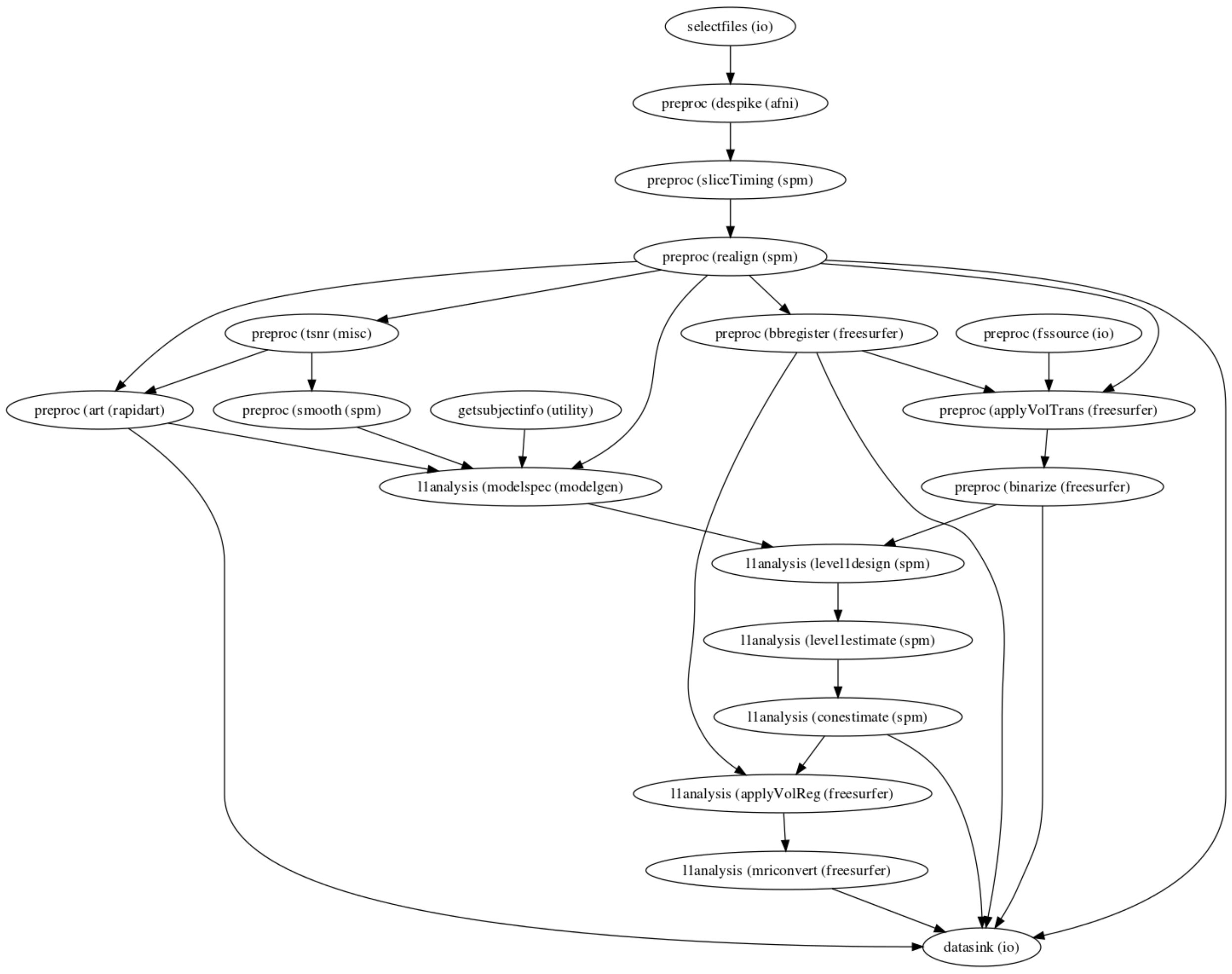
10.3.4 `flat` - detailed graph

The detailed graph of the visualization type `flat` shows the `metaflow` in all its glory. This graph shows all nodes, their inputs and outputs and how they are connected to each other.


infosource utility contrasts subject_id
selectfiles io subject_id func
getsubjectinfo utility subject_id subject_info
despike afni preproc in_file out_file
fssource io preproc subject_id brainmask
sliceTiming spm preproc in_files timecorrected_files
realign spm preproc in_files mean_image realigned_files realignment_parameters
bbregister freesurfer preproc source_file subject_id out_fsl_file out_reg_file registered_file
tsnr misc preproc in_file detrended_file
applyVolTrans freesurfer preproc reg_file source_file target_file transformed_file
smooth spm preproc in_files smoothed_files
art rapidart preproc mask_file realigned_files realignment_parameters outlier_files plot_files
modelspec modelgen l1analysis functional_runs outlier_files realignment_parameters subject_info session_info
binarize freesurfer preproc in_file binary_file
level1design spm l1analysis mask_image session_info spm_mat_file
level1estimate spm l1analysis spm_mat_file beta_images residual_image spm_mat_file
conestimate spm l1analysis beta_images contrasts residual_image spm_mat_file con_images spmT_images spm_mat_file
applyVolReg freesurfer l1analysis reg_file source_file transformed_file
mriconvert freesurfer l1analysis in_file out_file
datasink io bbregister.@out_fsl_file bbregister.@out_reg_file bbregister.@registered_file contrasts.@T contrasts.@con contrasts.@contrasts contrasts.@spm_mat functionals.@brainmask functionals.@mean functionals.@outliers functionals.@parameters functionals.@plot
IN OUT

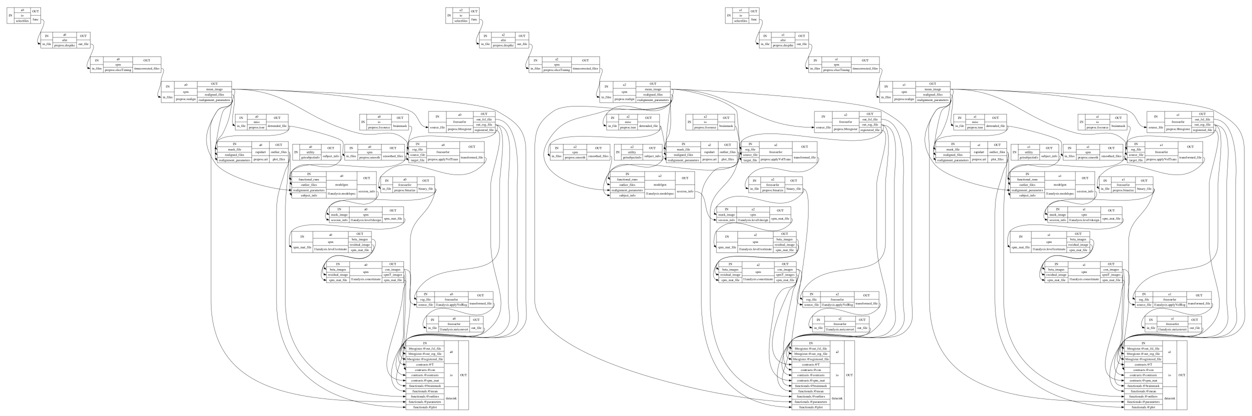
10.3.5 `exec` - simple graph

The detailed graph of the visualization type `exec` doesn't really show you anything different than the simple graph of the visualization type `flat`. The advantage of the `exec` type lies in the detailed graph.


selectfiles (io)
preproc (despike (afni)
preproc (sliceTiming (spm)
preproc (realign (spm)
preproc (tsnr (misc)
preproc (bbregister (freesurfer)
preproc (fssource (io)
preproc (art (rapidart)
preproc (smooth (spm)
getsubjectinfo (utility)
preproc (applyVolTrans (freesurfer)
l1analysis (modelspec (modelgen)
preproc (binarize (freesurfer)
l1analysis (level1design (spm)
l1analysis (level1estimate (spm)
l1analysis (conestimate (spm)
l1analysis (applyVolReg (freesurfer)
l1analysis (mriconvert (freesurfer)
datasink (io)

10.3.6 `exec` - detailed graph

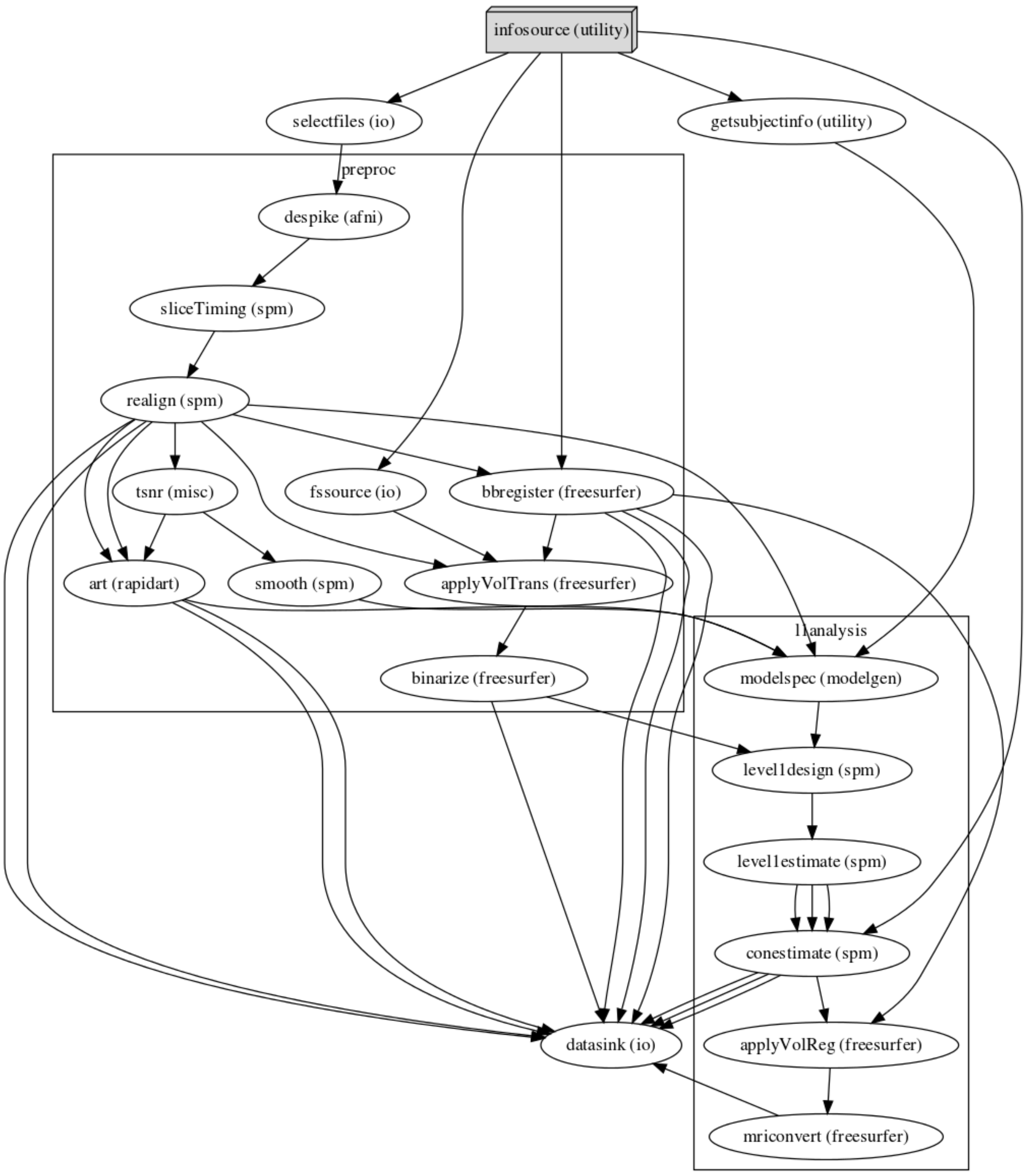
The detailed graph of the visualization type `exec` shows you the nodes of the `metaflow` with the same details as the visualization type `flat` would do. But additionally, all iterables are expanded so that you can see the full hierarchical and parallel structure of your analysis. In the following example the node `selectfiles` iterates over `sub001`, `sub002` and `sub003`.



Note: As you can see from this example, every iteration creates a subgraph with its own index. In this case `a0`, `a1` and `a2`. Such an indexing structure is also maintained in the folders and subfolders of your working and output directory.

10.3.7 `hierarchical` - simple graph

The graph of the visualization type `hierarchical` shows the `metaflow` as seen with the visualization type `flat` but emphasizes the hierarchical structure of its subworkflows. This is done by surrounding each subworkflow with a box labeled with the name of the subworkflow. Additionally, each node with an iterable field will be shown as a gray box.

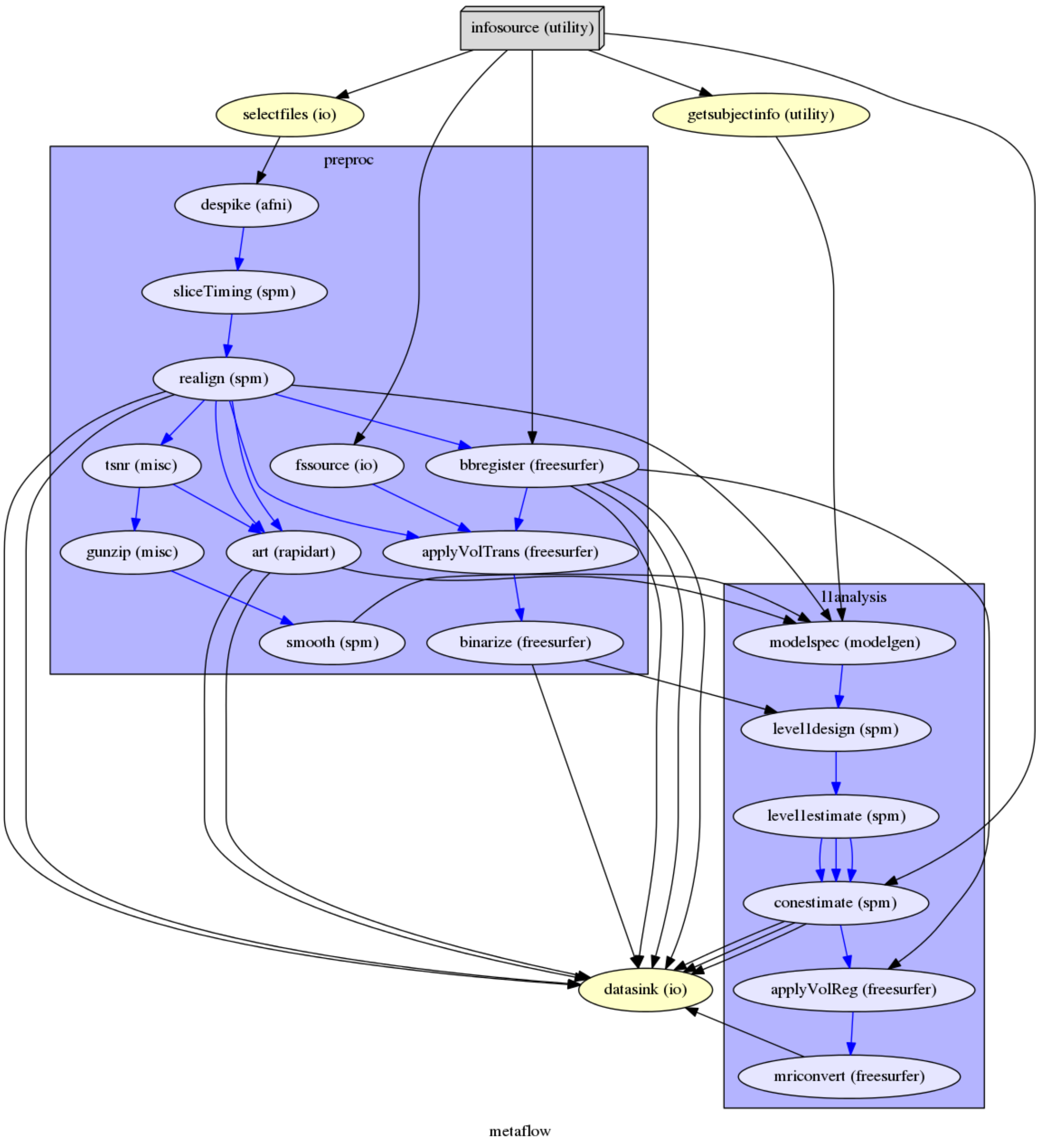


metaflow

In this example you see that the `metaflow` contains a `preproc` and a `l1analysis` workflow.

10.3.8 `colored` - simple graph

The graph of the visualization type `colored` shows the `metaflow` as seen with the visualization type `hierarchical` but color codes the different `hierarchical` levels as well as the connections between and within those levels with different colors.


infosource (utility)
selectfiles (io)
getsubjectinfo (utility)
preproc
despike (afni)
sliceTiming (spm)
realign (spm)
tsnr (misc)
fssource (io)
bbregister (freesurfer)
gunzip (misc)
art (rapidart)
applyVolTrans (freesurfer)
l1analysis
smooth (spm)
binarize (freesurfer)
modelspec (modelgen)
level1design (spm)
level1estimate (spm)
conestimate (spm)
datasink (io)
applyVolReg (freesurfer)
mriconvert (freesurfer)
metaflow

CHAPTER

ELEVEN

HOW TO BUILD A PIPELINE FOR A FIRST LEVEL FMRI ANALYSIS

In this section you will learn how to create a workflow that does a **first level analysis** on fMRI data. There are multiple ways how you can do this, as there are different ways and strategies to preprocess your data and different ways to create a model of your design. So keep in mind that the workflow in this section is just an example and may not suite your specific experiment or design. Having said that, it still contains the most common steps used in a first level analysis.

11.1 Define the structure of your pipeline

A typical first level fMRI workflow can be divided into two sections: **preprocessing** and **first level analysis**. The first one deals with removing noise and confounding factors such as movement from your data and the second one deals with fitting the model of your experiment to the data. The best way to build a pipeline from scratch is to think about the steps the data has to go through:

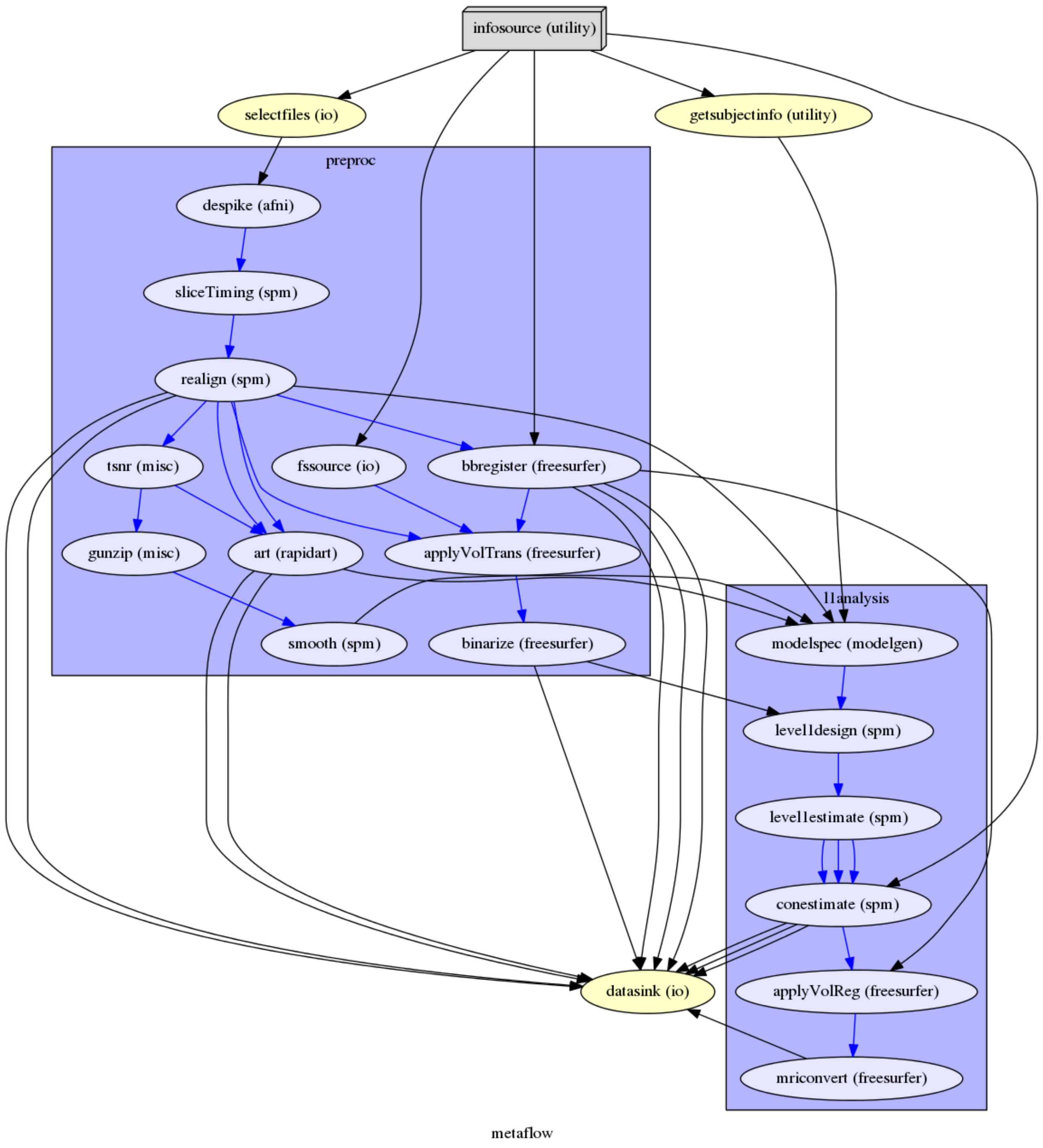
In the **preprocessing** part of the pipeline we first want to `Despike` the data. This is a process that removes local and short timeline 'spikes' from the functional data. After that we want to correct for the slice wise acquisition of the data with `SliceTiming` and get rid of the movement in the scanner with `Realign`. We want to detrend our data by removing polynomial to the 2nd order with `TSNR`. Additional to that, we also want to check the realignment parameters for extreme movements, i.e. artifacts, with `ArtifactDetect` and prepare an inclusion mask for the first level model with `DilateImage`. This inclusion mask is created by taking the `aseg.mgz` segmentation file from FreeSurfer (with the node `FreeSurferSource`), binarizing the values with `Binarize` (values above 0.5 become 1 and below become 0) and than dilating this binarized mask by a smoothing kernel with `DilateImage`. After all this is done we are ready to smooth our data with `Smooth`.

In the **first level analysis** part of the pipeline we first want to specify our model with `SpecifyModel`, than create the first level design with `Level1Design`. After that we are able to estimate the model with `EstimateModel` and estimate the contrasts with `EstimateContrast`. Before we're done we want to coregister our contrasts to the subject specific anatomy with `BBRegister` and `ApplyVolTransform` and convert the final output to zipped NIfTI files with `MRIConvert`.

As with every workflow, we also need some input and output nodes which handle the data and an additional node that provides subject specific information. For that we need an `Infosource` node that knows on which subjects to run, a `SelectFiles` node that knows where to get the data, a `getsubjectinfo` node that knows subject specific information (e.g. onset times and duration of conditions, subject specific regressors, etc.) and a `DataSink` node that knows which files to store and where to store them.

To run all this we need some structure. Therefore, we will put the **preprocessing** and the **first level analysis** part in its own subworkflow and create a hierarchically higher workflow that contains those two subworkflows as well as the input and output nodes. I'd like to call this top workflow **metaflow**.

Those are a lot of different parts and it is confusing to make sense of it without actually seeing what we try to build. Here is how the metaflow should look like in the end:



Note: The normalization of the first level output into a common reference space (e.g. MNI-space) will not be done by this metaflow. This because I want to dedicate a whole section on different ways on how to normalize your data (http://miykael.github.io/nipype-beginner-s-guide/normalize.html). Normalizing your data is very important for the analysis on the group level and a good normalization can be the difference between super results or none at all.

11.2 Write your pipeline script

Before we can start with writing a pipeline script, we first have to make sure that we have all necessary information. We know how the structure of the metaflow should look like from the previous section. But what are the experiment specific parameters? Lets assume that we use the tutorial dataset (http://miykael.github.io/nipype-beginner-s-guide/prepareData.html) with the ten subjects `sub001` to `sub010`, each having two functional scans `run001.nii.gz` and `run002.nii.gz`. We know from the openfmri homepage DS102: Flanker task (event-related) (https://openfmri.org/dataset/ds000102) that this experiment has a a TR of 2.0 seconds and that each volume of the functional data consists of 40 slices, acquired in an ascending interleaved slice order. And we know that we can find condition specific onset times for each subject in the subject folder in our data folder, e.g. `~/nipype_tutorial/data/sub001/`. So let's start!

11.2.1 Import modules

First we have to import all necessary modules. Which modules you have to import becomes clear while you're adding specific nodes.

```
from os.path import join as opj
from nipype.interfaces.afni import Despike
from nipype.interfaces.freesurfer import (BBRegister, ApplyVolTransform,
                                          Binarize, MRIConvert, FSCommand)
from nipype.interfaces.spm import (SliceTiming, Realign, Smooth, Level1Design,
                                   EstimateModel, EstimateContrast)
from nipype.interfaces.utility import Function, IdentityInterface
from nipype.interfaces.io import FreeSurferSource, SelectFiles, DataSink
from nipype.algorithms.rapidart import ArtifactDetect
from nipype.algorithms.misc import TSNR, Gunzip
from nipype.algorithms.modelgen import SpecifySPMModel
from nipype.pipeline.engine import Workflow, Node, MapNode
```

11.2.2 Specify interface behaviors

To make sure that the MATLAB and FreeSurfer interface run correctly, add the following code to your script.

```
# MATLAB - Specify path to current SPM and the MATLAB's default mode
from nipype.interfaces.matlab import MatlabCommand
MatlabCommand.set_default_paths('/usr/local/MATLAB/R2014a/toolbox/spm12')
MatlabCommand.set_default_matlab_cmd("matlab -nodesktop -nosplash")

# FreeSurfer - Specify the location of the freesurfer folder
fs_dir = '~/nipype_tutorial/freesurfer'
FSCommand.set_default_subjects_dir(fs_dir)
```

11.2.3 Define experiment specific parameters

I suggest to keep experiment specific parameters that change often between experiments like subject names, output folders, scan parameters and name of functional runs at the beginning of your script. Like this they can be accessed and changed more easily.

```
experiment_dir = '~/nipype_tutorial'          # location of experiment folder
subject_list = ['sub001', 'sub002', 'sub003',
                'sub004', 'sub005', 'sub006',
                'sub007', 'sub008', 'sub009',
                'sub010']                        # list of subject identifiers
output_dir = 'output_fMRI_example_1st'          # name of 1st-level output folder
working_dir = 'workingdir_fMRI_example_1st'    # name of 1st-level working directory

number_of_slices = 40                            # number of slices in volume
TR = 2.0                                         # time repetition of volume
fwhm_size = 6                                    # size of FWHM in mm
```

11.2.4 Create preprocessing pipeline

Let's first create all nodes needed for the preprocessing subworkflow:

```
# Despike - Removes 'spikes' from the 3D+time input dataset
despike = MapNode(Despike(outputtype='NIFTI'),
                  name="despike", iterfield=['in_file'])

# Slicetiming - correct for slice wise acquisition
interleaved_order = range(1,number_of_slices+1,2) + range(2,number_of_slices+1,2)
sliceTiming = Node(SliceTiming(num_slices=number_of_slices,
                               time_repetition=TR,
                               time_acquisition=TR-TR/number_of_slices,
                               slice_order=interleaved_order,
                               ref_slice=2),
                   name="sliceTiming")

# Realign - correct for motion
realign = Node(Realign(register_to_mean=True),
               name="realign")

# TSNR - remove polynomials 2nd order
tsnr = MapNode(TSNR(regress_poly=2),
               name='tsnr', iterfield=['in_file'])

# Artifact Detection - determine which of the images in the functional series
#   are outliers. This is based on deviation in intensity or movement.
art = Node(ArtifactDetect(norm_threshold=1,
                          zintensity_threshold=3,
                          mask_type='file',
                          parameter_source='SPM',
                          use_differences=[True, False]),
           name="art")

# Gunzip - unzip functional
gunzip = MapNode(Gunzip(), name="gunzip", iterfield=['in_file'])

# Smooth - to smooth the images with a given kernel
smooth = Node(Smooth(fwhm=fwhm_size),
              name="smooth")

# FreeSurferSource - Data grabber specific for FreeSurfer data
fssource = Node(FreeSurferSource(subjects_dir=fs_dir),
                run_without_submitting=True,
                name='fssource')

# BBRegister - coregister a volume to the Freesurfer anatomical
```

```
bbregister = Node(BBRegister(init='header',
                             contrast_type='t2',
                             out_fsl_file=True),
                  name='bbregister')

# Volume Transformation - transform the brainmask into functional space
applyVolTrans = Node(ApplyVolTransform(inverse=True),
                     name='applyVolTrans')

# Binarize -  binarize and dilate an image to create a brainmask
binarize = Node(Binarize(min=0.5,
                         dilate=1,
                         out_type='nii'),
                name='binarize')
```

After implementing the nodes we can create the preprocessing subworkflow and add all those nodes to it and connect them to each other.

```
# Create a preprocessing workflow
preproc = Workflow(name='preproc')

# Connect all components of the preprocessing workflow
preproc.connect([(despike, sliceTiming, [('out_file', 'in_files')]),
                 (sliceTiming, realign, [('timecorrected_files', 'in_files')]),
                 (realign, tsnr, [('realigned_files', 'in_file')]),
                 (tsnr, art, [('detrended_file', 'realigned_files')]),
                 (realign, art, [('mean_image', 'mask_file'),
                                 ('realignment_parameters',
                                  'realignment_parameters')]),
                 (tsnr, gunzip, [('detrended_file', 'in_file')]),
                 (gunzip, smooth, [('out_file', 'in_files')]),
                 (realign, bbregister, [('mean_image', 'source_file')]),
                 (fssource, applyVolTrans, [('brainmask', 'target_file')]),
                 (bbregister, applyVolTrans, [('out_reg_file', 'reg_file')]),
                 (realign, applyVolTrans, [('mean_image', 'source_file')]),
                 (applyVolTrans, binarize, [('transformed_file', 'in_file')]),
                 ])
```

If you are wondering how we know which parameters to specify and which connections to establish. It is simple: First, specify or connect all mandatory inputs of each node. Second, add the additional inputs that your data requires. For more informations about what is mandatory and what's not, go either to Interfaces and Algorithms (http://nipype.readthedocs.io/en/latest/interfaces/index.html) or use the `.help()` method (e.g. `realign.help()`), as shown here (http://miykael.github.io/nipype-beginner-s-guide/firstSteps.html#input-and-output-fields).

11.2.5 Create first level analysis pipeline

Now, let us define the pipeline for the first level analysis. Again, first we need to implement the nodes:

```
# SpecifyModel - Generates SPM-specific Model
modelspec = Node(SpecifySPMModel(concatenate_runs=False,
                                 input_units='secs',
                                 output_units='secs',
                                 time_repetition=TR,
                                 high_pass_filter_cutoff=128),
                 name="modelspec")

```

```
# Level1Design - Generates an SPM design matrix
level1design = Node(Level1Design(bases={'hrf': {'derivs': [0, 0]}},
                                 timing_units='secs',
                                 interscan_interval=TR,
                                 model_serial_correlations='AR(1)'),
                    name="level1design")

# EstimateModel - estimate the parameters of the model
level1estimate = Node(EstimateModel(estimation_method={'Classical': 1}),
                      name="level1estimate")

# EstimateContrast - estimates contrasts
conestimate = Node(EstimateContrast(), name="conestimate")

# Volume Transformation - transform contrasts into anatomical space
applyVolReg = MapNode(ApplyVolTransform(fs_target=True),
                      name='applyVolReg',
                      iterfield=['source_file'])

# MRIConvert - to gzip output files
mriconvert = MapNode(MRIConvert(out_type='niigz'),
                     name='mriconvert',
                     iterfield=['in_file'])
```

Now that this is done, we create the first level analysis subworkflow and add all the nodes to it and connect them to each other.

```
# Initiation of the 1st-level analysis workflow
l1analysis = Workflow(name='l1analysis')

# Connect up the 1st-level analysis components
l1analysis.connect([(modelspec, level1design, [('session_info',
                                                'session_info')]),
                    (level1design, level1estimate, [('spm_mat_file',
                                                     'spm_mat_file')]),
                    (level1estimate, conestimate, [('spm_mat_file',
                                                    'spm_mat_file'),
                                                   ('beta_images',
                                                    'beta_images'),
                                                   ('residual_image',
                                                    'residual_image')]),
                    (conestimate, applyVolReg, [('con_images',
                                                 'source_file')]),
                    (applyVolReg, mriconvert, [('transformed_file',
                                                'in_file')]),
                    ])
```

11.2.6 Define meta workflow and connect subworkflows

After we've created the subworkflows `preproc` and `l1analysis` we are ready to create the meta workflow `metaflow` and establish the connections between the two subworkflows.

```
metaflow = Workflow(name='metaflow')
metaflow.base_dir = opj(experiment_dir, working_dir)

metaflow.connect([(preproc, l1analysis, [('realign.realignment_parameters',
                                          'modelspec.realignment_parameters'),
                                         ('smooth.smoothed_files',
                                          'modelspec.functional_runs'),
                                         ('art.outlier_files',
                                          'modelspec.outlier_files'),
                                         ('binarize.binary_file',
                                          'level1design.mask_image'),
                                         ('bbregister.out_reg_file',
                                          'applyVolReg.reg_file'),
                                         ]),
                  ])
```

11.2.7 Define model specific parameters

The procedure of how we get subject specific parameters into our metaflow is a bit tricky but can be done as shown below. First, we have to specify the conditions of our paradigm and what contrasts we want to compute from them. In our case, the names of the condition are `'congruent'` and `'incongruent'`. The original condition of the tutorial dataset also include a subdivision into correct and incorrect trials (see `~/nipype_tutorial/data/condition_key.txt`). This example will not consider this subdivision, as there are very few or no occurrences of incorrect responses per subject.

```
# Condition names
condition_names = ['congruent', 'incongruent']

# Contrasts
cont01 = ['congruent',   'T', condition_names, [1, 0]]
cont02 = ['incongruent', 'T', condition_names, [0, 1]]
cont03 = ['congruent vs incongruent', 'T', condition_names, [1, -1]]
cont04 = ['incongruent vs congruent', 'T', condition_names, [-1, 1]]
cont05 = ['Cond vs zero', 'F', [cont01, cont02]]
cont06 = ['Diff vs zero', 'F', [cont03, cont04]]

contrast_list = [cont01, cont02, cont03, cont04, cont05, cont06]
```

The definition of contrasts is rather straight forward. For a T-contrast, just specify the name of the contrast, the type, the name of all conditions and the weights to those conditions. The implementation of an F-contrast only needs a name for the contrast, the type of the contrast, followed by a list of T-contrasts to use in the F-contrast. One important addition: If you want to have run specific contrasts add an additional list to the end of the contrast, which specifies for which run the contrast should be used. For example, if you want the 3rd contrast only computed in the 2nd run, use the following code:

```
cont03 = ['congruent', 'T', condition_names, [1, 0], [0, 1]]
```

Now let's get to the more tricky part: How do we get the subject and run specific onset times for the 'congruent' and the 'incongruent' condition into our pipeline? Well, with the following function:

```
# Function to get Subject specific condition information
def get_subject_info(subject_id):
    from os.path import join as opj
    path = '~/nipype_tutorial/data/%s'%subject_id
    onset_info = []
    for run in ['01', '02']:
        for cond in ['01', '02', '03', '04']:
            onset_file = opj(path, 'onset_run0%s_cond0%s.txt'%(run, cond))
            with open(onset_file, 'rt') as f:
                for line in f:
```

```
11                     info = line.strip().split()
12                     if info[1] != '0.00':
13                         onset_info.append(['cond0%s'%cond,
14                                            'run0%s'%run,
15                                            float(info[0])])
16     onset_run1_congruent = []
17     onset_run1_incongruent = []
18     onset_run2_congruent = []
19     onset_run2_incongruent = []
20
21     for info in onset_info:
22         if info[1] == 'run001':
23             if info[0] == 'cond001' or info[0] == 'cond002':
24                 onset_run1_congruent.append(info[2])
25             elif info[0] == 'cond003' or info[0] == 'cond004':
26                 onset_run1_incongruent.append(info[2])
27         if info[1] == 'run002':
28             if info[0] == 'cond001' or info[0] == 'cond002':
29                 onset_run2_congruent.append(info[2])
30             elif info[0] == 'cond003' or info[0] == 'cond004':
31                 onset_run2_incongruent.append(info[2])
32
33     onset_list = [sorted(onset_run1_congruent), sorted(onset_run1_incongruent),
34                   sorted(onset_run2_congruent), sorted(onset_run2_incongruent)]
35
36     from nipype.interfaces.base import Bunch
37     condition_names = ['congruent', 'incongruent']
38
39     subjectinfo = []
40     for r in range(2):
41         onsets = [onset_list[r*2], onset_list[r*2+1]]
42         subjectinfo.insert(r,
43                            Bunch(conditions=condition_names,
44                                  onsets=onsets,
45                                  durations=[[0], [0]],
46                                  amplitudes=None,
47                                  tmod=None,
48                                  pmod=None,
49                                  regressor_names=None,
50                                  regressors=None))
51     return subjectinfo
```

So what does it do? **Line 3 to 34** are specific to the tutorial dataset and will most certainly not apply for any other study, which are not from the openfmri.org (https://openfmri.org/). This part of the function goes through the subject folder under `~/nipype_tutorial/data/` and reads out the values in the files `onset_run00?_cond00?.txt`. The result of line 3 to 34 is an array called `onset_list` with four arrays, containing the onset for the condition `congruent_run1`, `incongruent_run1`, `congruent_run2` and `incongruent_run2`. In the case of `sub001` this looks like this:

```
onset_list=[[20.0, 30.0, 52.0, 64.0, 88.0, 116.0, 130.0, 140.0, 184.0, 196.0, 246.0,␣
↪274.0],
            [0.0, 10.0, 40.0, 76.0, 102.0, 150.0, 164.0, 174.0, 208.0, 220.0, 232.0,␣
↪260.0],
            [10.0, 20.0, 30.0, 42.0, 102.0, 116.0, 164.0, 174.0, 208.0, 220.0, 232.0,␣
↪260.0],
            [0.0, 54.0, 64.0, 76.0, 88.0, 130.0, 144.0, 154.0, 184.0, 196.0, 246.0,␣
↪274.0]]
```

Line 36 to 50 is the part of the `get_subject_info` function that has to be included in almost all first level analysis workflows. For more information see Model Specification for First Level fMRI Analysis (http://nipype.readthedocs.io/en/latest/users/model_specification.html). Important to know are the following things: The for loop `for r in range(2)` in line 40 is set to 2 because we have two runs per subject. The idea is to create an output variable `subjectinfo` that contains a `Bunch` object for each run. The content of this `Bunch` object depends on the subject and contains the name of the conditions, onsets of them, duration of each event, as well as possible amplitude modifications, temporal or polynomial derivatives or regressors. **Note:** The duration of all events per condition were set to `[0]`, as this assumes that the events should be modeled as impulses.

Now that the tricky part is done, we only need to create an additional node that applies this function and has the value of the `subjectinfo` variable as an output field. This can be done with a function node (as shown in the previous section (http://miykael.github.io/nipype-beginner-s-guide/firstSteps.html#individual-nodes))

```
# Get Subject Info - get subject specific condition information
getsubjectinfo = Node(Function(input_names=['subject_id'],
                               output_names=['subject_info'],
                               function=get_subject_info),
                      name='getsubjectinfo')
```

11.2.8 Establish Input & Output Stream

As always, our metaflow needs an input stream to have data to work and an output stream to know where to store the computed output. This can be done with the following three nodes:

- `infosource`: This node will iterate over the `subject_list` and feed the `contrast_list` to the first level analysis.
- `selectfiles`: This node will grab the functional files from the subject folder and feed them to the preprocessing pipeline, specifically the `Despike` node.
- `datasink`: This node will store the metaflow output in an output folder and rename or delete unwanted post- or prefixes.

And here's the code to do this:

```
# Infosource - a function free node to iterate over the list of subject names
infosource = Node(IdentityInterface(fields=['subject_id',
                                            'contrasts'],
                                    contrasts=contrast_list),
                  name="infosource")
infosource.iterables = [('subject_id', subject_list)]

# SelectFiles - to grab the data (alternativ to DataGrabber)
templates = {'func': 'data/{subject_id}/run*.nii.gz'}
selectfiles = Node(SelectFiles(templates,
                               base_directory=experiment_dir),
                   name="selectfiles")

# Datasink - creates output folder for important outputs
datasink = Node(DataSink(base_directory=experiment_dir,
                         container=output_dir),
                name="datasink")

# Use the following DataSink output substitutions
substitutions = [('_subject_id_', ''),
                 ('_despike', ''),
                 ('_detrended', ''),
                 ('_warped', '')]
```

```
datasink.inputs.substitutions = substitutions

# Connect Infosource, SelectFiles and DataSink to the main workflow
metaflow.connect([(infosource, selectfiles, [('subject_id', 'subject_id')]),
                  (infosource, preproc, [('subject_id',
                                          'bbregister.subject_id'),
                                         ('subject_id',
                                          'fssource.subject_id')]),
                  (selectfiles, preproc, [('func', 'despike.in_file')]),
                  (infosource, getsubjectinfo, [('subject_id', 'subject_id')]),
                  (getsubjectinfo, l1analysis, [('subject_info',
                                                 'modelspec.subject_info')]),
                  (infosource, l1analysis, [('contrasts',
                                             'conestimate.contrasts')]),
                  (preproc, datasink, [('realign.mean_image',
                                        'preprocout.@mean'),
                                       ('realign.realignment_parameters',
                                        'preprocout.@parameters'),
                                       ('art.outlier_files',
                                        'preprocout.@outliers'),
                                       ('art.plot_files',
                                        'preprocout.@plot'),
                                       ('binarize.binary_file',
                                        'preprocout.@brainmask'),
                                       ('bbregister.out_reg_file',
                                        'bbregister.@out_reg_file'),
                                       ('bbregister.out_fsl_file',
                                        'bbregister.@out_fsl_file'),
                                       ('bbregister.registered_file',
                                        'bbregister.@registered_file'),
                                       ]),
                  (l1analysis, datasink, [('mriconvert.out_file',
                                           'contrasts.@contrasts'),
                                          ('conestimate.spm_mat_file',
                                           'contrasts.@spm_mat'),
                                          ('conestimate.spmT_images',
                                           'contrasts.@T'),
                                          ('conestimate.con_images',
                                           'contrasts.@con'),
                                          ]),
                  ])
```

11.2.9 Run the pipeline and generate the graph

Finally, after everything is set up correctly we can run the pipeline and let it draw the graph of the workflow.

```
metaflow.write_graph(graph2use='colored')
metaflow.run('MultiProc', plugin_args={'n_procs': 8})
```

Hint: You can download the code for this first level pipeline as a script here: example_fMRI_1_first_level.py (https://github.com/miykael/nipype-beginner-s-guide/blob/master/scripts/example_fMRI_1_first_level.py)

11.3 Visualize your pipeline

The visualization of this graph can be seen in all different graph types under the section How to visualize a pipeline (http://miykael.github.io/nipype-beginner-s-guide/visualizePipeline.html) or as a colored graph at the beginning of this section.

11.4 Resulting Folder Structure

After we've run our **first level analysis pipeline** our folder structure should look like this:

After we've executed the first level workflow we have two new folders under `~/nipype_tutorial`. The working directory `workingdir_fMRI_example_1st` which contains all files created during the execution of the metaflow, and the output folder `output_fMRI_example_1st` which contains all the files that we sent to the DataSink. Let's take a closer look at the DataSink folder:

```
output_fMRI_example_1st
|-- bbregister
|   |-- sub001
|   |   |-- meanarun001_bbreg_sub001.dat
|   |   |-- meanarun001_bbreg_sub001.mat
|   |-- sub0..
|   |-- sub010
|-- contrasts
|   |-- sub001
|   |   |-- con_0001.nii
|   |   |-- con_0002.nii
|   |   |-- con_0003.nii
|   |   |-- con_0004.nii
|   |   |-- con_0005.nii
|   |   |-- ess_0005.nii
|   |   |-- ess_0006.nii
|   |   |-- _mriconvert0
|   |   |   |-- con_0001_out.nii.gz
|   |   |-- _mriconvert1
|   |   |   |-- con_0002_out.nii.gz
|   |   |-- _mriconvert2
|   |   |   |-- con_0003_out.nii.gz
|   |   |-- _mriconvert3
|   |   |   |-- con_0004_out.nii.gz
|   |   |-- _mriconvert4
|   |   |   |-- ess_0005_out.nii.gz
|   |   |-- _mriconvert5
|   |   |   |-- ess_0006_out.nii.gz
|   |   |-- spmF_0005.nii
|   |   |-- spmF_0006.nii
|   |   |-- SPM.mat
|   |   |-- spmT_0001.nii
|   |   |-- spmT_0002.nii
|   |   |-- spmT_0003.nii
|   |   |-- spmT_0004.nii
|   |-- sub0..
|   |-- sub010
|-- preprocout
    |-- sub001
    |   |-- art.rarun001_outliers.txt
    |    |-- art.rarun002_outliers.txt
    |    |-- brainmask_thresh.nii
    |    |-- meanarun001.nii
    |    |-- plot.rarun001.png
    |    |-- plot.rarun002.png
    |    |-- rp_arun001.txt
    |    |-- rp_arun002.txt
    |-- sub0..
    |-- sub010
```

The `bbregister` folder contains two files that both contain the registration information between the functional mean image and the anatomical image. The `.dat` file is the registration matrix in FreeSurfer and the `.mat` file in FSL format.

The `contrast` folder contains the estimated beta (`con` and `ess` files) and statistical spm (`spmT` and `spmF` files) contrasts. It also contains the `SPM.mat` file as well as 5 folders (`_mriconvert0` to `_mriconvert4`) which contain the coregistered and converted `con*_out.nii.gz` files.

The `preprocout` folder contains different informative and necessary output from the preprocess workflow:

- The `art.rarun00?_outliers.txt` files contain the number of outlier volumes, detected by the `ArtifactDetection` node.
- The `plot.rarun00?.png` images show the volume to volume change in intensity or movement, plotted by the `ArtifactDetection` node. Red vertical lines mean that the specified volume was detected as an outlier.
- The `rp_arun00?.txt` files contain the movement regressors calculated by the `Realign` node.
- The `brainmask_thresh.nii` file is the computed binary mask used in the `Level1Design` node.
- The file `meanarun001.nii` is the functional mean file computed by the `Realign` node.

CHAPTER

TWELVE

HOW TO NORMALIZE YOUR DATA

Before you can start with a second level analysis you are facing the problem that all your output from the first level analysis are still in their subject specific subject-space. Because of the huge differences in brain size and cortical structure, it is very important to transform the data of each subject from its individual subject-space into a common standardized reference-space. This process of transformation is what we call normalization and it consists of a rigid body transformation (translations and rotations) as well as of a affine transformation (zooms and shears). The most common template that subject data is normalized to is the MNI template (http://www.bic.mni.mcgill.ca/ServicesAtlases/HomePage).

There are many different approaches to when to normalize your data (e.g. before smoothing your data in preprocessing or after the estimation of your contrasts at the end of your first level analysis). I prefer to do the normalization after the first level model was estimated, as I don't want to introduce to many unnecessary transformations in the subject specific first level analysis.

There are also many different softwares that you could use to normalize your data and there is a lot of debate of which ones are better and which ones are worse. There is a very good paper by Klein et al. (2009), called Evaluation of 14 nonlinear deformation algorithms applied to human brain MRI registration (http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2747506/), that summarizes and compares the main approaches. According to this paper and from my own personal experience, I highly recommend to use ANTs (http://stnava.github.io/ANTs/) to normalize your data. I also recommend to not use SPM8's normalization module (to see why check out SPM12's release notes (http://www.fil.ion.ucl.ac.uk/spm/software/spm12/SPM12_Release_Notes.pdf#page=9), page 9, section "8.5. Old Normalise"). In contrary to SPM8, SPM12's new normalization approach seems to do very good.

This section will show you how to use Nipype to do a normalization with ANTs (what I recommend) or with SPM12. I highly recommend the normalization with ANTs as it is much more accurate. But I also want to point out that the computation of ANTs normalization is much longer than the one with SPM12, which also does a good enough job.

12.1 Normalize Your Data with ANTs

Usually, we first normalize the subject specific anatomy to the template and then use the resulting transformation matrix to normalize the functional data (i.e. first level contrasts) to the template. But this also means that we assume that the functional and the anatomical data is laying on top of each other, or in other words, that they were coregistered to each other.

The coregistration of the functional data to the anatomical data means multiple interpolation of your data. That's why the coregistration and the normalization of your functional data should be done directly in one transformation. To account for both possible cases, the following script will show you both ways. The approach where both coregistration and normalization will be done in one step will be called **complete transformation**, and the approach where we only do a normalization will be called **partial transformation**. Partial does not mean that it doesn't lead to a complete normalization of the data, it just means that the coregistration of the functional and structural data was already done in an earlier step.

Important: A big thanks to Julia Huntenburg (https://github.com/juhuntenburg)! Without her help and knowledge about ANTs and Nipype this section wouldn't exist, as she provided me with the relevant code and structure of this pipeline.

12.1.1 Import modules and specify interface behaviors

But first, as always, we have to import necessary modules and tell the system where to find the FreeSurfer folder.

```python
# Import modules
from os.path import join as opj
from nipype.interfaces.ants import Registration, ApplyTransforms
from nipype.interfaces.freesurfer import FSCommand, MRIConvert, BBRegister
from nipype.interfaces.c3 import C3dAffineTool
from nipype.interfaces.utility import IdentityInterface, Merge
from nipype.interfaces.io import SelectFiles, DataSink, FreeSurferSource
from nipype.pipeline.engine import Workflow, Node, MapNode
from nipype.interfaces.fsl import Info

# FreeSurfer - Specify the location of the freesurfer folder
fs_dir = '~/nipype_tutorial/freesurfer'
FSCommand.set_default_subjects_dir(fs_dir)
```

12.1.2 Define experiment specific parameters

Now we define the names of the folders used for this pipeline, we specify the list of subjects which should be normalized and specify which template the data should be normalized too. In this case it is the `MNI152_T1_1mm_brain.nii.gz` template.

```python
# Specify variables
experiment_dir = '~/nipype_tutorial'          # location of experiment folder
input_dir_1st = 'output_fMRI_example_1st'     # name of 1st-level output folder
output_dir = 'output_fMRI_example_norm_ants'  # name of norm output folder
working_dir = 'workingdir_fMRI_example_norm_ants'  # name of norm working directory
subject_list = ['sub001', 'sub002', 'sub003',
                'sub004', 'sub005', 'sub006',
                'sub007', 'sub008', 'sub009',
                'sub010']                     # list of subject identifiers

# location of template file
template = Info.standard_image('MNI152_T1_1mm_brain.nii.gz')
```

Hint: For other templates check out the ones in FSL's `standard` folder, in my case this is under `/usr/share/fsl/data/standard`. You can also see a list and access them much easier within Nipype with the following code:

```python
from nipype.interfaces.fsl import Info
Info.standard_image()
```

12.1.3 Create nodes

In both cases, the **complete** as well as the **partial** transformation approach, we will use ANTs' `Registration` to compute the transformation matrix between the subject specific anatomy and the template:

```
# Registration (good) - computes registration between subject's structural and MNI␣
↪template.
antsreg = Node(Registration(args='--float',
                            collapse_output_transforms=True,
                            fixed_image=template,
                            initial_moving_transform_com=True,
                            num_threads=1,
                            output_inverse_warped_image=True,
                            output_warped_image=True,
                            sigma_units=['vox']*3,
                            transforms=['Rigid', 'Affine', 'SyN'],
                            terminal_output='file',
                            winsorize_lower_quantile=0.005,
                            winsorize_upper_quantile=0.995,
                            convergence_threshold=[1e-06],
                            convergence_window_size=[10],
                            metric=['MI', 'MI', 'CC'],
                            metric_weight=[1.0]*3,
                            number_of_iterations=[[1000, 500, 250, 100],
                                                  [1000, 500, 250, 100],
                                                  [100, 70, 50, 20]],
                            radius_or_number_of_bins=[32, 32, 4],
                            sampling_percentage=[0.25, 0.25, 1],
                            sampling_strategy=['Regular',
                                               'Regular',
                                               'None'],
                            shrink_factors=[[8, 4, 2, 1]]*3,
                            smoothing_sigmas=[[3, 2, 1, 0]]*3,
                            transform_parameters=[(0.1,),
                                                  (0.1,),
                                                  (0.1, 3.0, 0.0)],
                            use_histogram_matching=True,
                            write_composite_transform=True),
               name='antsreg')
```

This registration node `antsreg` might take a while, depending on the power of your system. One way to speed up this process is by using multiple cores/threads for the calculation. For example, use 4 cores for the calculation by setting the parameter `num_threads` to 4. But be aware, that if you run the normalization pipeline for 10 subjects in parallel, the code will try to launch 10 instances of ANTs registration with each asking for 4 cores.

Another approach to reduce the computation time of the registration is by reducing its accuracy by changing the parameters of `Registration` according to the following script: https://github.com/stnava/ANTs/blob/master/Scripts/newAntsExample.sh to the following:

```
# Registration (fast) - computes registration between subject's structural and MNI␣
↪template.
antsregfast = Node(Registration(args='--float',
                                collapse_output_transforms=True,
                                fixed_image=template,
                                initial_moving_transform_com=True,
                                num_threads=1,
                                output_inverse_warped_image=True,
                                output_warped_image=True,
                                sigma_units=['vox']*3,
                                transforms=['Rigid', 'Affine', 'SyN'],
                                terminal_output='file',
                                winsorize_lower_quantile=0.005,
                                winsorize_upper_quantile=0.995,
                                convergence_threshold=[1e-08, 1e-08, -0.01],
                                convergence_window_size=[20, 20, 5],
                                metric=['Mattes', 'Mattes', ['Mattes', 'CC']],
                                metric_weight=[1.0, 1.0, [0.5, 0.5]],
                                number_of_iterations=[[10000, 11110, 11110],
                                                      [10000, 11110, 11110],
                                                      [100, 30, 20]],
                                radius_or_number_of_bins=[32, 32, [32, 4]],
                                sampling_percentage=[0.3, 0.3, [None, None]],
                                sampling_strategy=['Regular',
                                                   'Regular',
                                                   [None, None]],
                                shrink_factors=[[3, 2, 1],
                                                [3, 2, 1],
                                                [4, 2, 1]],
                                smoothing_sigmas=[[4.0, 2.0, 1.0],
                                                  [4.0, 2.0, 1.0],
                                                  [1.0, 0.5, 0.0]],
                                transform_parameters=[(0.1,),
                                                      (0.1,),
                                                      (0.2, 3.0, 0.0)],
                                use_estimate_learning_rate_once=[True]*3,
                                use_histogram_matching=[False, False, True],
                                write_composite_transform=True),
                   name='antsregfast')
```

Now that we have the transformation matrix to normalize the functional data to the template, we can use ANTs' `ApplyTransforms` to execute that. **Note**: Here you have again the option to specify the number of threads used in the interpolation of the data.

Partial Transformation

In the partial transformation approach, we only need the following additional nodes. One to normalize the anatomical and one to normalize the functional data.

```
# Apply Transformation - applies the normalization matrix to contrast images
apply2con = MapNode(ApplyTransforms(args='--float',
                                    input_image_type=3,
                                    interpolation='Linear',
                                    invert_transform_flags=[False],
                                    num_threads=1,
                                    reference_image=template,
                                    terminal_output='file'),
                    name='apply2con', iterfield=['input_image'])

# Apply Transformation - applies the normalization matrix to the mean image
apply2mean = Node(ApplyTransforms(args='--float',
                                  input_image_type=3,
                                  interpolation='Linear',
                                  invert_transform_flags=[False],
                                  num_threads=1,
                                  reference_image=template,
                              terminal_output='file'),
                  name='apply2mean')
```

Complete Transformation

For the complete transformation, we also need to calculate the coregistration matrix (we will use FreeSurfer's *BBRegister* for that). But first, we need to use FreeSurfer's `FreeSurferSource` to grab the subject specific anatomy, convert it from MGZ to NII format with `MRIConvert`. Than we need to transform the BBRegister transformation matrix to ITK format with `C3dAffineTool` and merge this transformation matrix with the transformation matrix from the normalization, i.e. `antsreg`, by using a `Merge` node.

```
# FreeSurferSource - Data grabber specific for FreeSurfer data
fssource = Node(FreeSurferSource(subjects_dir=fs_dir),
                run_without_submitting=True,
                name='fssource')

# Convert FreeSurfer's MGZ format into NIfTI format
convert2nii = Node(MRIConvert(out_type='nii'), name='convert2nii')

# Coregister the median to the surface
bbregister = Node(BBRegister(init='fsl',
                             contrast_type='t2',
                             out_fsl_file=True),
                  name='bbregister')

# Convert the BBRegister transformation to ANTS ITK format
convert2itk = Node(C3dAffineTool(fsl2ras=True,
                                 itk_transform=True),
                   name='convert2itk')


# Concatenate BBRegister's and ANTS' transforms into a list
merge = Node(Merge(2), iterfield=['in2'], name='mergexfm')
```

Note: Before you can use the `C3dAffineTool` you have to make sure that you have the C3D routines on your system. Otherwise you get the following error: `IOError: c3d_affine_tool could not be found on host`. To download the newest C3D version, go to this homepage (http://sourceforge.net/projects/c3d/). Afterwards, unpack and install the code on your system, this can be done with the following command: `sudo tar xzvf ~/Downloads/c3d-nightly-Linux-x86_64.tar.gz -C /usr/local/.`. And finally, to make sure that your system finds the binaries of this software, add the following line to your `.bashrc` file: `export PATH=/usr/local/c3d-1.0.0-Linux-x86_64/bin:$PATH`.

Now that we have the couplet transformation matrix, we can normalize the anatomical and functional data with the following two nodes:

```
# Transform the contrast images. First to anatomical and then to the target
warpall = MapNode(ApplyTransforms(args='--float',
                                  input_image_type=3,
                                  interpolation='Linear',
                                  invert_transform_flags=[False, False],
                                  num_threads=1,
                                  reference_image=template,
                                  terminal_output='file'),
                  name='warpall', iterfield=['input_image'])
```

```

# Transform the mean image. First to anatomical and then to the target
warpmean = Node(ApplyTransforms(args='--float',
                                input_image_type=3,
                                interpolation='Linear',
                                invert_transform_flags=[False, False],
                                num_threads=1,
                                reference_image=template,
                                terminal_output='file'),
                name='warpmean')
```

Note: A very important difference between the partial and the complete approach is that the parameter `invert_transform_flags` of the `ApplyTransforms` has two values in the case where we have two transformation matrices and only one value where we have only one transformation. If this is not accounted for than you get the following error: `ERROR: The useInverse list must have the same number of entries as the transformsFileName list.`

12.1.4 Create the pipeline and connect nodes to it

```
# Initiation of the ANTS normalization workflow
normflow = Workflow(name='normflow')
normflow.base_dir = opj(experiment_dir, working_dir)
```

For the **partial transformation** use the following code:

```
# Connect up ANTS normalization components
normflow.connect([(antsreg, apply2con, [('composite_transform', 'transforms')]),
                  (antsreg, apply2mean, [('composite_transform',
                                          'transforms')])
                  ])
```

For the **complete transformation** use the following code:

```
# Connect up ANTS normalization components
normflow.connect([(fssource, convert2nii, [('T1', 'in_file')]),
                  (convert2nii, convert2itk, [('out_file', 'reference_file')]),
                  (bbregister, convert2itk, [('out_fsl_file',
                                              'transform_file')]),
                  (convert2itk, merge, [('itk_transform', 'in2')]),
                  (antsreg, merge, [('composite_transform',
                                     'in1')]),
                  (merge, warpmean, [('out', 'transforms')]),
                  (merge, warpall, [('out', 'transforms')]),
                  ])
```

12.1.5 Establish Input & Output Stream

```
# Infosource - a function free node to iterate over the list of subject names
infosource = Node(IdentityInterface(fields=['subject_id']),
                  name="infosource")
infosource.iterables = [('subject_id', subject_list)]

```

```
# SelectFiles - to grab the data (alternativ to DataGrabber)
anat_file = opj('freesurfer', '{subject_id}', 'mri/brain.mgz')
func_file = opj(input_dir_1st, 'contrasts', '{subject_id}',
                '_mriconvert*/*_out.nii.gz')
func_orig_file = opj(input_dir_1st, 'contrasts', '{subject_id}', '[ce]*.nii')
mean_file = opj(input_dir_1st, 'preprocout', '{subject_id}', 'mean*.nii')

templates = {'anat': anat_file,
             'func': func_file,
             'func_orig': func_orig_file,
             'mean': mean_file,
             }

selectfiles = Node(SelectFiles(templates,
                               base_directory=experiment_dir),
                   name="selectfiles")

# Datasink - creates output folder for important outputs
datasink = Node(DataSink(base_directory=experiment_dir,
                         container=output_dir),
                name="datasink")

# Use the following DataSink output substitutions
substitutions = [('_subject_id_', ''),
                 ('_apply2con', 'apply2con'),
                 ('_warpall', 'warpall')]
datasink.inputs.substitutions = substitutions
```

For the **partial transformation** use the following code:

```
# Connect SelectFiles and DataSink to the workflow
normflow.connect([(infosource, selectfiles, [('subject_id', 'subject_id')]),
                  (selectfiles, apply2con, [('func', 'input_image')]),
                  (selectfiles, apply2mean, [('mean', 'input_image')]),
                  (selectfiles, antsreg, [('anat', 'moving_image')]),
                  (antsreg, datasink, [('warped_image',
                                        'antsreg.@warped_image'),
                                       ('inverse_warped_image',
                                        'antsreg.@inverse_warped_image'),
                                       ('composite_transform',
                                        'antsreg.@transform'),
                                       ('inverse_composite_transform',
                                        'antsreg.@inverse_transform')]),
                  (apply2con, datasink, [('output_image',
                                          'warp_partial.@con')]),
                  (apply2mean, datasink, [('output_image',
                                           'warp_partial.@mean')]),
                  ])
```

For the **complete transformation** use the following code:

```
# Connect SelectFiles and DataSink to the workflow
normflow.connect([(infosource, selectfiles, [('subject_id', 'subject_id')]),
                  (infosource, fssource, [('subject_id', 'subject_id')]),
                  (infosource, bbregister, [('subject_id', 'subject_id')]),
                  (selectfiles, bbregister, [('mean', 'source_file')]),
                  (selectfiles, antsreg, [('anat', 'moving_image')]),
                  (selectfiles, convert2itk, [('mean', 'source_file')]),
                  (selectfiles, warpall, [('func_orig', 'input_image')]),
                  (selectfiles, warpmean, [('mean', 'input_image')]),
                  (antsreg, datasink, [('warped_image',
                                        'antsreg.@warped_image'),
                                       ('inverse_warped_image',
                                        'antsreg.@inverse_warped_image'),
                                       ('composite_transform',
                                        'antsreg.@transform'),
                                       ('inverse_composite_transform',
                                        'antsreg.@inverse_transform')]),
                  (warpall, datasink, [('output_image', 'warp_complete.@warpall')]),
                  (warpmean, datasink, [('output_image', 'warp_complete.@warpmean')]),
                  ])
```

12.1.6 Run the pipeline and generate the graphs

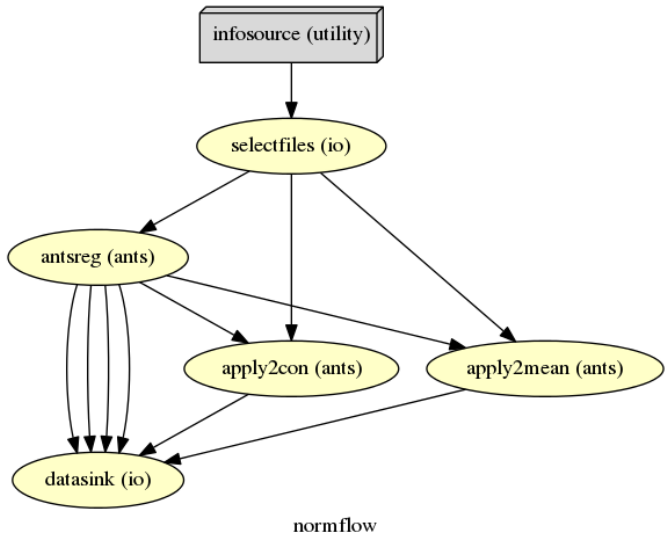
Now, let's run the workflow with the following code:

```
normflow.write_graph(graph2use='colored')
normflow.run('MultiProc', plugin_args={'n_procs': 8})
```

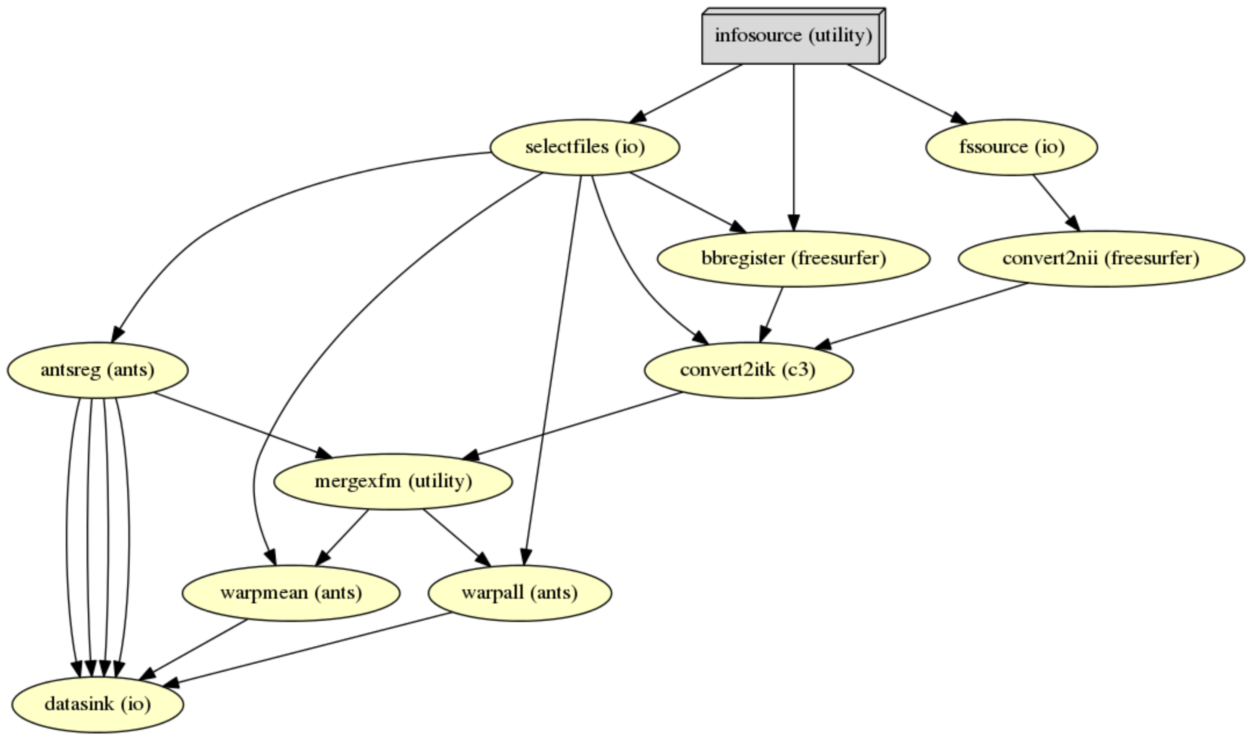
Hint: You can download the code for the partial and complete normalization with ANTS as a script here: example_fMRI_2_normalize_ANTS_complete.py (https://github.com/miykael/nipype-beginner-s-guide/blob/master/scripts/example_fMRI_2_normalize_ANTS_complete.py) or example_fMRI_2_normalize_ANTS_partial.py (https://github.com/miykael/nipype-beginner-s-guide/blob/master/scripts/example_fMRI_2_normalize_ANTS_partial.py)

12.1.7 Visualize the workflow

The colored graph of the **partial** normalization workflow looks as follows:



The colored graph of the **complete** normalization workflow looks as follows:



normflow

12.1.8 Resulting Folder Structure

The resulting folder structure looks for the **partial** and **complete** approach combined as follows:

```
output_fMRI_example_norm_ants/
|-- antsreg
|   |-- sub001
|   |   |-- transformComposite.h5
|   |   |-- transformInverseComposite.h5
|   |   |-- transform_InverseWarped.nii.gz
|   |   |-- transform_Warped.nii.gz
|   |-- sub0..
|   |-- sub010
|-- warp_partial
|   |-- sub001
|   |   |-- apply2con0
|   |   |   |-- con_0001_out_trans.nii.gz
|   |   |-- apply2con1
|   |   |   |-- con_0002_out_trans.nii.gz
|   |   |-- apply2con2
|   |   |   |-- con_0003_out_trans.nii.gz
|   |   |-- apply2con3
|   |   |   |-- con_0004_out_trans.nii.gz
|   |   |-- apply2con4
|   |   |   |-- ess_0005_out_trans.nii.gz
|   |   |-- apply2con5
|   |   |   |-- ess_0006_out_trans.nii.gz
|   |   |-- meanarun001_trans.nii
|   |-- sub0..
|   |-- sub010
```

```
|-- warp_complete
    |-- sub001
    |    |-- meanarun001_trans.nii
    |    |-- warpall0
    |    |    |-- con_0001_trans.nii
    |    |-- warpall1
    |    |    |-- con_0002_trans.nii
    |    |-- warpall2
    |    |    |-- con_0003_trans.nii
    |    |-- warpall3
    |    |    |-- con_0004_trans.nii
    |    |-- warpall4
    |    |    |-- ess_0005_trans.nii
    |    |-- warpall5
    |         |-- ess_0006_trans.nii
    |-- sub0..
    |-- sub010
```

12.2 Normalize Your Data with SPM12

The normalization of your data with SPM12 is much simpler than the one with ANTs. We only need to feed all the necessary inputs to a node called `Normalize12`.

12.2.1 Import modules and specify interface behaviors

As always, let's import necessary modules and tell the system where to find MATLAB.

```python
# Import modules
from os.path import join as opj
from nipype.interfaces.spm import Normalize12
from nipype.interfaces.utility import IdentityInterface
from nipype.interfaces.io import SelectFiles, DataSink
from nipype.algorithms.misc import Gunzip
from nipype.pipeline.engine import Workflow, Node, MapNode

# Specification to MATLAB
from nipype.interfaces.matlab import MatlabCommand
MatlabCommand.set_default_paths('/usr/local/MATLAB/R2014a/toolbox/spm12')
MatlabCommand.set_default_matlab_cmd("matlab -nodesktop -nosplash")
```

12.2.2 Define experiment specific parameters

```python
# Specify variables
experiment_dir = '~/nipype_tutorial'          # location of experiment folder
input_dir_1st = 'output_fMRI_example_1st'     # name of 1st-level output folder
output_dir = 'output_fMRI_example_norm_spm'   # name of norm output folder
working_dir = 'workingdir_fMRI_example_norm_spm'   # name of working directory
subject_list = ['sub001', 'sub002', 'sub003',
                'sub004', 'sub005', 'sub006',
                'sub007', 'sub008', 'sub009',
                'sub010']                     # list of subject identifiers

```

```
# location of template in form of a tissue probability map to normalize to
template = '/usr/local/MATLAB/R2014a/toolbox/spm12/tpm/TPM.nii'
```

It's important to note that SPM12 provides its own template `TMP.nii` to which the data will be normalized to.

12.2.3 Create nodes

The functional and anatomical data that we want to normalize is in compressed ZIP format, which SPM12 can't handle. Therefore we first have to unzip those files with `Gunzip`, before we can feed those files to SPM's `Normalize12` node.

```
# Gunzip - unzip the structural image
gunzip_struct = Node(Gunzip(), name="gunzip_struct")

# Gunzip - unzip the contrast image
gunzip_con = MapNode(Gunzip(), name="gunzip_con",
                     iterfield=['in_file'])

# Normalize - normalizes functional and structural images to the MNI template
normalize = Node(Normalize12(jobtype='estwrite',
                             tpm=template,
                             write_voxel_sizes=[1, 1, 1]),
                 name="normalize")
```

12.2.4 Create the pipeline and connect nodes to it

```
# Specify Normalization-Workflow & Connect Nodes
normflow = Workflow(name='normflow')
normflow.base_dir = opj(experiment_dir, working_dir)

# Connect up ANTS normalization components
normflow.connect([(gunzip_struct, normalize, [('out_file', 'image_to_align')]),
                  (gunzip_con, normalize, [('out_file', 'apply_to_files')]),
                  ])
```

12.2.5 Establish Input & Output Stream

```
# Infosource - a function free node to iterate over the list of subject names
infosource = Node(IdentityInterface(fields=['subject_id']),
                  name="infosource")
infosource.iterables = [('subject_id', subject_list)]

# SelectFiles - to grab the data (alternativ to DataGrabber)
anat_file = opj('data', '{subject_id}', 'struct.nii.gz')
con_file = opj(input_dir_1st, 'contrasts', '{subject_id}',
                        '_mriconvert*/*_out.nii.gz')
templates = {'anat': anat_file,
             'con': con_file,
             }
selectfiles = Node(SelectFiles(templates,
                               base_directory=experiment_dir),
                   name="selectfiles")
```

```

# Datasink - creates output folder for important outputs
datasink = Node(DataSink(base_directory=experiment_dir,
                         container=output_dir),
                name="datasink")

# Use the following DataSink output substitutions
substitutions = [('_subject_id_', '')]
datasink.inputs.substitutions = substitutions

# Connect SelectFiles and DataSink to the workflow
normflow.connect([(infosource, selectfiles, [('subject_id', 'subject_id')]),
                  (selectfiles, gunzip_struct, [('anat', 'in_file')]),
                  (selectfiles, gunzip_con, [('con', 'in_file')]),
                  (normalize, datasink, [('normalized_files',
                                          'normalized.@files'),
                                         ('normalized_image',
                                          'normalized.@image'),
                                         ('deformation_field',
                                          'normalized.@field'),
                                         ]),
                  ])
```

12.2.6 Run the pipeline and generate the graphs

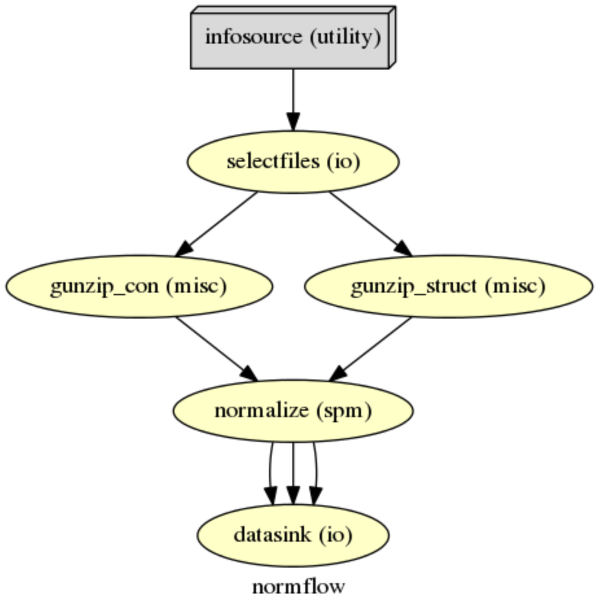
Now, let's run the workflow with the following code:

```
normflow.write_graph(graph2use='colored')
normflow.run('MultiProc', plugin_args={'n_procs': 8})
```

Hint: You can download the code for the normalization with SPM12 as a script here: example_fMRI_2_normalize_SPM.py (https://github.com/miykael/nipype-beginner-s-guide/blob/master/scripts/example_fMRI_2_normalize_SPM.py)

12.2.7 Visualize the workflow

The colored graph of this normalization workflow looks as follows:



12.2.8 Resulting Folder Structure

The resulting folder structure looks as follows:

```
output_fMRI_example_norm_spm/
|-- normalized
    |-- sub001
    |   |-- wcon_0001_out.nii
    |   |-- wcon_0002_out.nii
    |   |-- wcon_0003_out.nii
    |   |-- wcon_0004_out.nii
    |   |-- wess_0005_out.nii
    |   |-- wess_0006_out.nii
    |   |-- wstruct.nii
    |   |-- y_struct.nii
    |-- sub0..
    |-- sub010
```

CHAPTER

THIRTEEN

HOW TO BUILD A PIPELINE FOR A SECOND LEVEL FMRI ANALYSIS

In this section you will learn how to create a workflow that does a **second level analysis** on fMRI data. There are again multiple ways how you can do this, but the most simple on is to check if your contrasts from the first level analysis are still significant on the group-level a.k.a. the 2nd level.

Note: You can only do a **second level analysis** if you already have done a first level analysis, obviously! But more importantly, those first level contrasts have to be in a common reference space. Otherwise there is now way of actually comparing them with each other and getting a valid results from them. Luckily, if you've done the previous step, you've already normalized your data (either with ANTs or SPM) to a template.

13.1 Write your pipeline script

If you've already done the previous sections, you know how this works. We first import necessary modules, define experiment specific parameters, create nodes, create a workflow and connect the nodes to it, we create an I/O stream and connect it to the workflow and finally run this workflow.

13.1.1 Import modules and specify interface behaviors

```python
from os.path import join as opj
from nipype.interfaces.io import SelectFiles, DataSink
from nipype.interfaces.spm import (OneSampleTTestDesign, EstimateModel,
                                   EstimateContrast, Threshold)
from nipype.interfaces.utility import IdentityInterface
from nipype.pipeline.engine import Workflow, Node

# Specification to MATLAB
from nipype.interfaces.matlab import MatlabCommand
MatlabCommand.set_default_paths('/usr/local/MATLAB/R2014a/toolbox/spm12')
MatlabCommand.set_default_matlab_cmd("matlab -nodesktop -nosplash")
```

13.1.2 Define experiment specific parameters

```python
experiment_dir = '~/nipype_tutorial'             # location of experiment folder
output_dir = 'output_fMRI_example_2nd_ants'      # name of 2nd-level output folder
input_dir_norm = 'output_fMRI_example_norm_ants'# name of norm output folder
working_dir = 'workingdir_fMRI_example_2nd_ants'# name of working directory
subject_list = ['sub001', 'sub002', 'sub003',
                'sub004', 'sub005', 'sub006',
                'sub007', 'sub008', 'sub009',
                'sub010']                       # list of subject identifiers
contrast_list = ['con_0001', 'con_0002', 'con_0003',
                 'con_0004', 'ess_0005', 'ess_0006'] # list of contrast identifiers
```

Note: Pay attention to the name of the `input_dir_norm`. Depending on the way you normalized your data, ANTs or SPM, the folder name has either the ending `_ants` or `_spm`.

13.1.3 Create nodes

We don't need many nodes for a simple second level analysis. In fact they are the same as the ones we used for the first level analysis. We create a simple T-Test, estimate it and look at a simple mean contrast, i.e. a contrast that shows what the group mean activation of a certain first level contrast is.

```
# One Sample T-Test Design - creates one sample T-Test Design
onesamplettestdes = Node(OneSampleTTestDesign(),
                         name="onesampttestdes")

# EstimateModel - estimate the parameters of the model
level2estimate = Node(EstimateModel(estimation_method={'Classical': 1}),
                      name="level2estimate")

# EstimateContrast - estimates simple group contrast
level2conestimate = Node(EstimateContrast(group_contrast=True),
                         name="level2conestimate")
cont1 = ['Group', 'T', ['mean'], [1]]
level2conestimate.inputs.contrasts = [cont1]
```

13.1.4 Create the pipeline and connect nodes to it

```
# Specify 2nd-Level Analysis Workflow & Connect Nodes
l2analysis = Workflow(name='l2analysis')
l2analysis.base_dir = opj(experiment_dir, working_dir)

# Connect up the 2nd-level analysis components
l2analysis.connect([(onesamplettestdes, level2estimate, [('spm_mat_file',
                                                          'spm_mat_file')] ),
                    (level2estimate, level2conestimate, [('spm_mat_file',
                                                          'spm_mat_file'),
                                                         ('beta_images',
                                                          'beta_images'),
                                                         ('residual_image',
                                                          'residual_image')]),
                    ])
```

13.1.5 Establish Input & Output Stream

The creation of the I/O stream is as usual. But because I showed you three ways to normalize your data in the previous section, be aware that you have to point the `SelectFiles` node to the right input folder. Your option for the `SelectFiles` input template are as follows:

```
# contrast template for ANTs normalization (complete)
con_file = opj(input_dir_norm, 'warp_complete', 'sub*', 'warpall*',
               '{contrast_id}_trans.nii')

# contrast template for ANTs normalization (partial)
con_file = opj(input_dir_norm, 'warp_partial', 'sub*', 'apply2con*',
               '{contrast_id}_out_trans.nii.gz')

# contrast template for SPM normalization
con_file = opj(input_dir_norm, 'normalized', 'sub*',
               '*{contrast_id}_out.nii')
```

Note: It is very important to notice that only contrast images (e.g. `con`-images) can be used for a second-level group analysis. It is statistically incorrect to use statistic images, such as `spmT`- or `spmF`-images.

The following example is adjusted for the situation where the normalization was done with ANTs. The code for the I/O stream looks as follows:

```
# Infosource - a function free node to iterate over the list of subject names
infosource = Node(IdentityInterface(fields=['contrast_id']),
                  name="infosource")
infosource.iterables = [('contrast_id', contrast_list)]

# SelectFiles - to grab the data (alternative to DataGrabber)
con_file = opj(input_dir_norm, 'warp_complete', 'sub*', 'warpall*',
               '{contrast_id}_trans.nii')
templates = {'cons': con_file}

selectfiles = Node(SelectFiles(templates,
                               base_directory=experiment_dir),
                   name="selectfiles")

# Datasink - creates output folder for important outputs
datasink = Node(DataSink(base_directory=experiment_dir,
                         container=output_dir),
                name="datasink")

# Use the following DataSink output substitutions
substitutions = [('_contrast_id_', '')]
datasink.inputs.substitutions = substitutions

# Connect SelectFiles and DataSink to the workflow
l2analysis.connect([(infosource, selectfiles, [('contrast_id',
                                                 'contrast_id')]),
                    (selectfiles, onesamplettestdes, [('cons', 'in_files')]),
                    (level2conestimate, datasink, [('spm_mat_file',
                                                     'contrasts.@spm_mat'),
                                                    ('spmT_images',
                                                     'contrasts.@T'),
                                                    ('con_images',
                                                     'contrasts.@con')]),
                    ])
```

If you've normalized your data with ANTs but did only the so called **partial** approach, the code above will not work and crash with the following message:

```
Item 'Scans', field 'val': Number of matching files (0) less than required (1).

Standard error:
MATLAB code threw an exception:
...
Name:pyscript_onesamplettestdesign
...
Interface OneSampleTTestDesign failed to run.
```

Such errors are sometimes hard to read. What this message means is that SPM's `onesamplettestdes` tried to open an image-file but was only able to read out 0 scans, of the requested at least 1. This is a common message where SPM tries to read a zipped NIfTI file (ending with `nii.gz`) and cannot unpack it. To solve this issue we only need to insert an additional `Gunzip` node in our pipeline and redirect the workflow through this new gunzip node before it goes to the `onesamplettestdes` node. So the new code looks as follows:

```python
# Gunzip - unzip the contrast image
from nipype.algorithms.misc import Gunzip
from nipype.pipeline.engine import MapNode
gunzip_con = MapNode(Gunzip(), name="gunzip_con",
                     iterfield=['in_file'])

# Connect SelectFiles and DataSink to the workflow
l2analysis.connect([(infosource, selectfiles, [('contrast_id',
                                                'contrast_id')]),
                    (selectfiles, gunzip_con, [('cons', 'in_file')]),
                    (gunzip_con, onesamplettestdes, [('out_file',
                                                      'in_files')]),
                    (level2conestimate, datasink, [('spm_mat_file',
                                                    'contrasts.@spm_mat'),
                                                   ('spmT_images',
                                                    'contrasts.@T'),
                                                   ('con_images',
                                                    'contrasts.@con')]),
                    ])
```

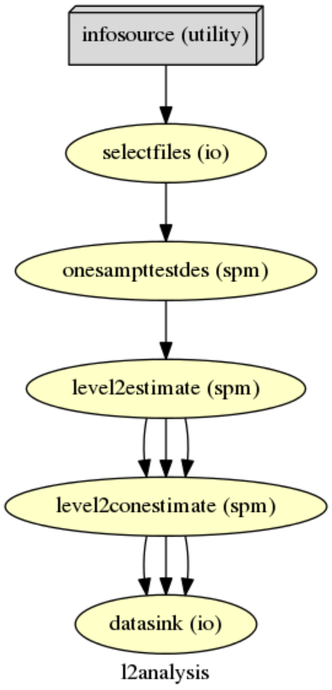
13.1.6 Run the pipeline and generate the graph

```python
l2analysis.write_graph(graph2use='colored')
l2analysis.run('MultiProc', plugin_args={'n_procs': 8})
```

Hint: You can download the code for this 2nd level pipeline as a script here: example_fMRI_3_second_level.py (https://github.com/miykael/nipype-beginner-s-guide/blob/master/scripts/example_fMRI_3_second_level.py)

13.2 Visualize your pipeline

The colored graph of the 2nd-level workflow looks as follows:



13.3 Resulting Folder Structure

The resulting folder structure looks as follows:

```
output_fMRI_example_2nd
|-- contrasts
    |-- con_0001
    |   |-- con_0001.nii
    |   |-- SPM.mat
    |   |-- spmT_0001.nii
    |-- con_0002
    |-- con_0003
    |-- con_0004
    |-- ess_0005
    |   |-- ess_0005.nii
    |   |-- SPM.mat
    |   |-- spmF_0005.nii
    |-- ess_0006
```

CHAPTER

FOURTEEN

HELP & SUPPORT

14.1 How Nipype Can Help You

Often times the first trouble with Nipype arise because of misunderstanding a node or its function. This can be because a mandatory input was forgotten, a input or output field is not what you thought it was or something similar. That's why the first step when running into a problem while building a pipeline should be to check out the description of the interface that causes the trouble. I've already described this here (https://miykael.github.io/nipype-beginner-s-guide/firstSteps.html#nodes) but to recap:

Let's assume that you've imported FreeSurfer's BBRegister with the command `from nipype.interfaces.freesurfer import BBRegister`. Now, if you want to know what this module generally does, use the *?* character, i.e. `BBRegister?`. This gives you a short description as well as an implementation example of this module. If you want to know a more detailed description of BBRegister, with all mandatory and possible inputs and outputs, use the `help()` function, i.e. `BBRegister.help()`.

Also, I highly recommend to check out Nipype's official Documentation (http://nipype.readthedocs.io/en/latest/documentation.html) section, where you can browse through all possible interfaces, function and description of them.

14.2 How to Help Yourself

If you have any **questions about or comments to this beginner's guide**, don't hesitate to leave a comment on the bottom of the corresponding homepage (you don't need an account to do so) or contact me directly by e-mail under: miykaelnotter@gmail.com.

If you have general **questions about what certain neuroimaging or nipype term** mean, check out the beginner's guide Glossary (https://miykael.github.io/nipype-beginner-s-guide/glossary.html) section.

If you have any **questions about Nipype or neuroimaging** itself please go directly to neurostars.org (https://neurostars.org/) or the beginner's guide FAQ (https://miykael.github.io/nipype-beginner-s-guide/faq.html) section. Neurostars.org (https://neurostars.org/) is a community driven Q&A platform that will help you to answer any nipype or neuroimaging related question that you possibly could have.

14.3 How to Help Me

The list of interfaces Nipype supports grows everyday more and more and the best practice to analyze MRI data is changing all the time. It's impossible for one person to keep track of all of those softwares and to know the state of the art analysis. That's why I'm very much counting on the input and support of the community to help me to make this beginner's guide as detailed and complete as possible.

So, if you found any mistakes, want to point out some alternative ways to do something or have any scripts or tutorials to share, your input is highly appreciated!

The best way to help me is to fork my repo on github (https://github.com/miykael/nipype-beginner-s-guide/tree/master/homepage) and send me a pull request. Alternatively you can also contact me with your ideas or feedback under miykaelnotter@gmail.com.

14.4 How to Read Crash Files

Everytime Nipype crashes, it creates a nice crash file containing all necessary information. For a specific example see this section (https://miykael.github.io/nipype-beginner-s-guide/firstSteps.html#common-issues-problems-and-crashes). In this example the name of the crash file is `crash-20141018-140440-mnotter-art.b0.pklz`.

The name of the file gives you already an information about when it crashed (`20141018-140440`) and which node crashed (`art.b0`). If you want to read the node you can use the terminal command `nipype_display_crash`. In our example the command to read the crash file is:

```
nipype_display_crash ~/nipype_tutorial/crash-20141018-140857-mnotter-art.b0.pklz
```

This leads to the following output:

```
File: crash-20141018-140857-mnotter-art.b0.pklz
Node: preproc.art.b0
Working directory: ~/nipype_tutorial/workingdir_firstSteps/
                     preproc/_subject_id_sub001/art

Node inputs:

bound_by_brainmask = False
global_threshold = 8.0
ignore_exception = False
intersect_mask = <undefined>
mask_file = <undefined>
mask_threshold = <undefined>
mask_type = spm_global
norm_threshold = 0.5
parameter_source = SPM
plot_type = png
realigned_files = <undefined>
realignment_parameters = ['~/nipype_tutorial/workingdir_firstSteps/preproc/
                             _subject_id_sub001/realign/rp_arun001.txt']
rotation_threshold = <undefined>
save_plot = True
translation_threshold = <undefined>
use_differences = [True, False]
use_norm = True
zintensity_threshold = 3.0

Traceback (most recent call last):
  File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/plugins/linear.py",
       line 38, in run node.run(updatehash=updatehash)
  File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/engine.py",
       line 1424, in run self._run_interface()
  File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/engine.py",
       line 1534, in _run_interface self._result = self._run_command(execute)
  File "~/anaconda/lib/python2.7/site-packages/nipype/pipeline/engine.py",
       line 1660, in _run_command result = self._interface.run()
  File "~/anaconda/lib/python2.7/site-packages/nipype/interfaces/base.py",
       line 965, in run self._check_mandatory_inputs()
  File "~/anaconda/lib/python2.7/site-packages/nipype/interfaces/base.py",
       line 903, in _check_mandatory_inputs raise ValueError(msg)
ValueError: ArtifactDetect requires a value for input 'realigned_files'.
            For a list of required inputs, see ArtifactDetect.help()
```

The first part of the crash report contains information about the node and the second part contains the error log. In this example, the last two lines tell us exactly, that the crash was caused because the input field *'realigned_files'* was not specified. This error is easy corrected, just add the required input and rerun the workflow.

Under certain circumstances it is possible and desired to rerun the crashed node. This can be done with the additional command flags to `nipype_display_crash`. The following flags are available:

```
-h, --help         show this help message and exit
-r, --rerun        rerun crashed node (default False)
-d, --debug        enable python debugger when re-executing (default False)
-i, --ipydebug     enable ipython debugger when re-executing (default False)
--dir DIRECTORY    Directory to run the node in (default None)
```

Note: For more information about how to debug your code and handle crashes go to this official Nipype section (http://nipype.readthedocs.io/en/latest/users/debug.html).

CHAPTER

FIFTEEN

LINKS TO NIPYPE, PROGRAMMING AND NEUROIMAGING

15.1 Nipype

- Nipype (http://nipype.readthedocs.io/en/latest/): The official homepage contains all about Nipype that you want to know.
- Michael Waskom (https://github.com/mwaskom/nipype_concepts) has written very nice introductions about Workflows (http://nbviewer.jupyter.org/github/mwaskom/nipype_concepts/blob/master/workflows.ipynb), Interfaces (http://nbviewer.jupyter.org/github/mwaskom/nipype_concepts/blob/master/interfaces.ipynb) and Iteration (http://nbviewer.jupyter.org/github/mwaskom/nipype_concepts/blob/master/iteration.ipynb).
- A short and very nice introduction to Nipype, written by by Satra Ghosh, can be found here (http://nbviewer.jupyter.org/github/nipy/nipype/blob/master/examples/nipype_tutorial.ipynb).

15.2 Programming: Python, Git & more

I highly recommend to use Sublime Text 3 (http://www.sublimetext.com/3) to write and edit your scripts. It's a really good and cool looking text editor with many helpful things, like the Anaconda Plugion (http://damnwidget.github.io/anaconda/) for example.

15.2.1 Learn Python

- A Byte of Python (http://python.swaroopch.com/): A very nice introduction to Python in general.
- A Crash Course in Python for Scientists (http://nbviewer.jupyter.org/gist/rpmuller/5920182): a very good introduction to Python and scientific programming (e.g. Numpy, Scipy, Matplotlib)
- Codecademy - Python (https://www.codecademy.com/learn/python): An interactive online training and introduction to Python.
- Learn Python the Hard Way (http://learnpythonthehardway.org/book/index.html): A very good step by step introduction to Python.
- Python Scientific Lecture Notes (http://www.scipy-lectures.org/): A very good and more detailed introduction to Python and scientific programming.
- If you're looking for a Python based IDE like Eclipse or MATLAB, check out Pycharm (https://www.jetbrains.com/pycharm/) or Spyder (https://github.com/spyder-ide/spyder/).
- Programming with Python (http://swcarpentry.github.io/python-novice-inflammation/): This short introduction by *software carpentry* teaches you the basics of scientific programming on very practical examples.

15.2.2 Learn Git

- Got 15 minutes and want to learn Git? (https://try.github.io/levels/1/challenges/1): Github's own git tutorial. It's fun and very short.
- Git Real (http://gitreal.codeschool.com/) on Code School (https://www.codeschool.com/): An interactive tutorial about GIT
- Top 10 Git Tutorials for Beginners (http://sixrevisions.com/resources/git-tutorials-beginners/)

15.2.3 Learn Unix Shell

- the Unix Shell (http://swcarpentry.github.io/shell-novice/): If you're new to Linux, here's a quick starter guide by software carpentry that teaches you the basics.

15.2.4 Learn Programming

- A very good article that outlines what programming languages are and where they are coming from can be found on IT Hare's Beginners Guide to Programming Languages (http://ithare.com/a-beginners-guide-to-programming-languages/). A big thank you to Sarah and the class of Mrs. Lowe from Colorado for recommending this site to me!
- If you want to learn more about other programming languages such as HTML & CSS, Javascript, jQuery, PHP, Python and Ruby, check out Beginner's Resources to Learn Programming Languages (https://www.vodien.com/blog/education/beginners-resources-to-learn-programming-languages.php). A big thank you to Mr. Tom Coner and his group for recommending the site to me. It's a great starting point to dive into any programming language.

15.3 Neuroimaging

- Neurostars.org (https://neurostars.org/): If you have any questions about Neuroinformatics, this is the place to go!
- Design efficiency in FMRI (http://imaging.mrc-cbu.cam.ac.uk/imaging/DesignEfficiency): A nice and detailed guide on how to design a good fMRI study.

CHAPTER

SIXTEEN

FREQUENTLY ASKED QUESTIONS

This FAQ wasn't created by myself. Its content is almost exclusively from the Imaging Knowledge Base - FAQ (http://mindhive.mit.edu/node/46) from the Gabrieli Lab at MIT (http://gablab.mit.edu/). It is my thanks to them, for generating such an exhaustive FAQ.

Index of FAQ

Note: To help me to extend this FAQ, please feel free to leave a comment at the bottom to recommend additions and to clear misapprehension.

16.1 Basic Statistical Modeling

16.1.1 1. What is "estimating a model?" How do the various programs do it?

Once you've performed all the spatial preprocessing you like on your functional data, you're ready to test your hypotheses about the data. Most standard analysis pathways in fMRI proceed in a hypothesis-driven fashion based on the general linear model (GLM), in which the researcher sets up a model of what she believes the brain may be doing in response to the variations in some parameter of the experiment, and then tests the truth of that hypothesis. (This contrasts with non-model-driven approaches like principal components analysis (PCA). Estimating your model is the core statistical step of this process: the researcher describes some model of brain activity, and the program calculates a giant multiple regression of some kind to find out the extent to which that model correctly accounts for the real data, at every voxel in the brain.

Different programs have different methods of setting up a design matrix, but they all share certain elements: the user describes a set of different experimental conditions (or effects), and describes, for each of them, start times and end times (onset times and offset times, or durations). Experiments can have massively varying designs, from the simplest on-off block design to multi-condition randomly-timed event-related design. The basic hypothesis is that some voxels in the brain had their intensity values covary, to a statistically significant degree, with some combination of the experimental conditions and parameters. The design matrix consists of a matrix with a row for each timepoint in the experiment (each functional image), and a column for each modeled experimental effect.

Usually, the user will then modify the design matrix to make it a more accurate model of what brain activity might be. Oftentimes, a constant term is added to the matrix, to account for the mean value of the session; sometimes linear or polynomial drifts are added to the matrix as well. Sometimes the columns of the matrix are convolved with some model of the hemodynamic response function, to reflect the blurring in signal the HRF applies to neural activity. (Another option is to simply separate the various timepoints for the response to a given condition into different columns, estimating each separately - a finite impulse response (FIR) model that effectively deconvolves the contribution of the HRF.) (See *HRF* for more info.)

Once the design matrix is set up, the program uses the methods of GLM theory - essentially multiple regression - to calculate how accurately the model described by the design matrix accounts for the real data. The standard GLM equation is $Y = BX + E$, where Y is the time-varying intensities from one voxel, X is the design matrix, E is an error term, and B is the "parameters" or "beta weights" - a vector of values, one for each experimental condition, that tells the researcher how big the effect of the corresponding condition was in explaining the values at that voxel. If condition A's beta weight is significantly greater than condition B's beta weight at a given voxel, the hypothesis that A had a greater effect than B at that voxel is confirmed. Generally, programs create some voxel-by-voxel image of the beta weights - a beta image or parameter image.

Once the parameters are estimated, the program has both a measure of effect size and of error in the model for each voxel. Generally, the program then normalizes each effect size by the error to calculate some measure of statistical significance for effects - a contrast image (see *Contrasts* for more info).

16.1.2 2. When should global mean scaling be used? What does it do?

Nutshell answer - global mean scaling should be used for PET, but not for fMRI.

Longer answer: One problem in neuroimaging experiments is that you're generally trying to pick out some signal from a noisy timeseries at every voxel. One form that noise can take is a global shift in intensities across the whole brain, which can be caused by scanner thermal noise, subject movement, physiological effects, etc. One way to get rid of a whole bunch of those noise sources at once, then, would be to look for timepoints where every voxel in the brain shows the same sudden shift and infer that that's a change in global response, not a regional change, and therefore not of interest to you. A simple way of doing that is just by finding the global mean of every timepoint - a global mean timeseries - and dividing every voxel's timeseries by the global mean timeseries.

The obvious problem with this is that removing the effect of the global mean from your model means you also remove signal that covaries with the global mean. In PET, this wasn't a big deal - the changes in the global mean could be on a different order of magnitude from task-related changes, and so regional activations weren't likely to bias the global mean particularly. In fMRI, though, it's a problem. Global mean intensity shifts are often about the same size, or at least comparable, to the size of task-induced activations. So large activations can seriously bias the global mean calculation, such that the global mean will go significantly up with large activations. Removing the effect of the global mean will then remove those large activations as well.

Using global scaling has been tested fairly extensively now in fMRI, and it almost always seems to negatively affect the sensitivity of the analysis. Generally, it's a bad idea.

16.1.3 3. What is autocorrelation correction? Should I do it?

The GLM approach suggested above and used for many types of experiments (not just neuroimaging) has a problem when applied to fMRI. It assumes that each observation is independent of the next, and any noise present at one timepoint is uncorrelated from the noise at the next timepoint. On that assumption, calculating the degrees of freedom in the data is easy - it's just the number of rows in the design matrix (number of TRs) minus the number of columns (number of effects), which makes calculating statistical significance for any beta value easy as well.

The trouble with this assumption is that it's wrong for fMRI. The large bulk of the noise present in the fMRI signal is low-frequency noise, which is highly correlated from one timepoint to the next. From a spectral analysis point of view, the power spectrum of the noise isn't flat by a long shot - it's highly skewed to the low frequency. In other words, there is a high degree of autocorrelation in fMRI data - the value at each time point is significantly explained by the value at the timepoints before or after. This is a problem for estimating statistical significance, because it means that our naive calculation of degrees of freedom is wrong - there are fewer degrees of freedom in real life than if every timepoint were independent, because of the high level of correlation between time points. Timepoints don't vary completely freely - they are explained by the previous timepoints. So our effective degrees of freedom is smaller than our earlier guess - but in order to calculate how significant any beta value is, we need to know how much smaller. How can we do that?

Friston and Worsley made an early attempt at. They argued that one way to account for the unknown autocorrelation was to essentially wash it out by applying their own, known, autocorrelation - temporally smoothing (or low-pass filtering) the data. If you extend the GLM framework to incorporate a known autocorrelation function and correctly calculate effective degrees of freedom for temporally smoothed data. This approach is sometimes called "coloring" the data - since uncorrelated noise is called "white" noise, this smoothing essentially "colors" the noise by rendering it less white. The idea is that after coloring, you know what color you've imposed, and so you can figure out exactly how to account for the color.

SPM99 (and earlier) offer two forms of accounting for the autocorrelation - low-pass filtering and autocorrelation estimation (AR(1) model). The autocorrelation estimation corresponds more with pre-whitening, although it's implemented badly in SPM99 and probably shouldn't be used. In practice, however, low-pass filtering seems to be a failure. Tests of real data have repeatedly shown that temporal smoothing of the data seems to hurt analysis sensitivity more than it helps, and harm false-positive rates more than it helps. The bias in fMRI noise is simply so significant that it can't be swamped without accounting for it. In real life, the proper theoretical approach seems to be pre-whitening, and low-pass filtering has been removed from SPM2 and continues to not be available in other major packages. (See *Temporal Filtering* for more info.)

16.1.4 4. What is pre-whitening? How does it help?

The other approach to dealing with autocorrelation in the fMRI noise power spectrum, instead of 'coloring' the noise, is to 'whiten' it. If the GLM assumes white noise, the argument runs, let's make the noise we really have into white noise. This is generally how correlated noise is dealt with in the GLM literature, and it can be shown whitening the noise gives the most unbiased parameter estimates possible. The way to do this is simply by running a regreession on your data to find the extent of the autocorrelation. If you can figure out how much each timepoint's value is biased by the one before it, you can remove the effect of that previous timepoint, and that way only leave the 'white' part of the noise.

In theory, this can be very tricky, because one doesn't actually know how many previous timepoints are influencing the current timepoint's value. Essentially, one is trying to model the noise, without having precise estimates of where the noise is coming from. In practice, however, enough work has been done on figuring out the sources of fMRI noise to have a fairly good model of what it looks like, and an AR(1) + w model, where each noise timepoint is some white noise plus a scaling of the noise timepoint before it, seems to be a good fit (it's also described as a 1/f model). This procedure essentially estimates the level of autocorrelation (or 'color') in the noise, and removes it from the timeseries ('whitening' the noise).

Theoretically, it should work well, but as its adoption is relatively new to the field, few rigorous tests of the effectiveness of pre-whitening have been done.

16.1.5 5. How does parametric modulation work? When would I use it?

As described above, there are all kind of modifications the researcher can make to her design matrix once she's described the basics of when her conditions are happening. One important one is parametric modulation, which can be used in a case where an experimental condition is not just ON or OFF, but can happen at a variety of levels during the experiment. An example might be an n-back memory task, where on each trial the subject is asked to remember what letter happened n trials before, where n is varied from trial to trial. One hypothesis the research might have is that activity in the brain varies as a function of n - remembering back 3 trials is harder than remembering 1, so you might expect activity on a 3-back trial to be higher than on a 1-back. In this case, a parametric modulation of the design matrix would be perfect.

Generally, a parametric modulation is useful if you have some numerical value for each trial that you'd like to model. This contrasts with having a numerical value to model at each timepoint, which would be a time for a user-specified regressor. In the parametric case, the user specifies onset times for the condition, and then specifies a parameter value for each trial in the condition - if there are 10 n-back trials, the user specifies 10 parameter values. The design matrix then modulates the activity in that column for each trial by some function of the parameter - linear, exponential, polynomial, etc. - set by the user. If the hypothesis is correct, that modulated column will fit the activity significantly better than an unmodulated effect.

16.1.6 6. What's the best way to include reaction times in my model?

If you have events for which participants' response times vary widely (or even a little), your model will be improved by accounting for this variation (rather than assuming all events take identical time, as in the normal model). A common way of including reaction times is to use a parametric modulator, with the reaction time for each trial included as the parameter. In the most common way of doing this, the height of the HRF will be thus modulated by the reaction time. Grinband et al. (HBM06) showed this method actually doesn't work as well as a different kind of parametric regression - in which each event is modeled as an epoch (i.e., a boxcar) of variable duration, convolved with a standard HRF.

In other words, rather than assuming that neural events all take the same time, and the HRF they're convolved by varies in height with reaction time (not very plausible, or, it turns out, efficient), the best way is to assume the underlying neural events vary in reaction time, and convolve those boxcars (rather than "stick functions") with the same HRF.

In either case, as with most parametric modulation, the regressor including reaction time effects can be separate from the "trial regressor" that models the reaction-time-invariant effect of the trial. This corresponds to having one column in the design matrix for the condition itself (which doesn't have any reaction time effects) and a second, parametrically modulated one, which includes reaction times. If your goal is merely to get the best model possible, these don't need to be separated (only the second of the two, which includes RTs, could go in the model), but this will not allow you to separate the effect of "just being in the trial" from neural activations that vary with reaction time. To separate those effects, you need separate design matrix columns to model them. That choice depends on how interested you are in the reaction-time effect itself.

16.1.7 7. What kinds of user-specified regressors might I use? How do I include them?

Another modification you can make to the design matrix is simply to add columns or effects that don't correspond to some condition you want convolved with an HRF. A user-specified regressor is just some vector of numbers, one for each timepoint/functional image, that you'd like to include in the model because you believe it has some effect. If you have a numerical value for each timepoint (TR/functional image) that you'd like to model, a user-specified regressor is the way to go. This contrasts with the case of having a numerical value for each trial you'd like to model, in which case you'd use a parametric modulation.

An example of a user-specified regressor might be if you have continuous self-reports of positive affect from each subject, and you'd like to see where there are voxels in the brain whose activity co-varied with that affect. You could include the positive affect regressor in your model and have a beta value estimated separately for it. Depending on what your hypothesis is about that effect, you may want to lag its values to account for the hemodynamic delay.

The user-specified regressor is a powerful tool for many types of modifications to the design matrix, but note that in many obvious cases in which you might want to separate out the contribution of a given effect of no interest - things like movement parameters, physiological variation, low-frequency confounds, etc. - programs may already have ways to deal with those things built in, in a more efficient fashion. At the very least, in any case when you include a user-specified regressor than you plan to simply ignore, you should try to ensure it doesn't covary significantly with your task and hence remove task-induced signal.

16.2 Connectivity

16.2.1 1. What is functional connectivity? What is effective connectivity?

The concept of "brain connectivity" is, as Horwitz points out, rather a tricky one to define. Ideally, you'd like to be able to measure the spatial (and temporal) path that information follows, from one point to another, millisecond by millisecond, and neuron to neuron (or at least region to region), in a directed fashion, such that you could say, "Ah, yes, activation starts in the visual cortex, moves to V2, gets shuttled from there to these other three visual areas and parietal cortex, and from parietal there to this other bit." Then you'd know something about what was being calculated and what calculations were being done where (and when). But, of course, you can't do that (yet). In fact, in general, most neuroscience recording methods, be they single-cell recording or fMRI or anything else, deal with isolated units of analysis - single cells or single voxels. You can't, in general, really well measure one neuron's connection to another in a living, behaving animal, much less noninvasively in a person.

What you can do is sample several sites at once and try and see how the patterns of activity you get are connected to each other. An obvious pattern to look for would be if two sites/voxels have intensities that are highly correlated. If the timeseries from one voxel looks exactly like the timeseries from another voxel, it might be a good bet they're doing similar things. If they're right next to each other, you call it a cluster; if they're far away from each - say, in visual cortex and PFC - you might guess they're connected to each other somehow.

Trouble is, of course, you run into the old adage that correlation doesn't imply causation. High correlation between remote sampling sites might imply some direct connection, or it might imply some third site driving their joint activation, or it might imply them jointly driving some third site. And even if they are connected, it's difficult to tell the direction of the connection, even if there is a "direction" to it.

Hence the two different terms used to describe connectivity in neuroimaging, a split introduced by Friston in 1993. Functional connectivity is the correlation concept - it's a descriptive concept, simply defined as the temporal correlation between remote timeseries or samples or what have you. Finding functional connectivity essentially reduces, as Lee et. al point out, to finding whether activity in two regions share any mutual information or not. Effective connectivity, by contrast, is the causation concept. It's defined as "the influence one neural system exerts over another either directly or indirectly." It doesn't imply a direct physical connection - simply a causative influence. It's a concept meant to support explanation and inference, more than just description, and it requires some account of causative direction or why there isn't any. It's also a lot trickier to figure out, generally, than functional connectivity. You'll hear both terms tossed around a fair amount, but remember: functional is simply correlation, whereas effective requires some causation somewhere.

16.2.2 2. How do I measure connectivity in the brain?

Good question. Almost every method for functional neuroimaging has ways to measure connectivity, and almost all of them boil down to the same concept: measuring the connection between timeseries at different points in the brain. In other words, almost every method out there measures functional connectivity, rather than directly measuring effective connectivity. Whether you're doing EEG and correlating timeseries from different electrodes, or using the fanciest dynamic causal modeling mathematical strategy with fast-TR fMRI, you're restricted generally to the data you can measure, which are samples from voxels that are treated independently. You can rule out some possible directions of influence by rules like temporal precedence (if a spike in one area precedes one from another area, the latter area can't have caused the earlier spike), but in general, most connectivity measures work on this simple foundation: Sample timeseries of activity from many different areas (voxels, electrodes, etc.) and then mathematically derive some measure of the mutual information between selected timeseries.

There are a couple obvious exceptions to this foundation. Measuring anatomical connectivity is a different type of procedure, and it's not clear how much influence anatomical connectivity (as we can measure it) and functional connecitivity have with each other - or should have with each other in analysis. Lee et. al has an intriguing discussion on this point (and many others). At some level, of course, if we're interested in finding out whether information is flowing from one neuron to another, it's useful to know if they're directly connected or not. But we're a long way off from those sorts of measures on a large scale, and it's not clear that coarser measures are all that useful in learning about functional connectivity. Anatomical connectivity on its own can be incredibly interesting, though, which is why diffusion tensor imaging (DTI) is becoming increasingly popular as an imaging modality. The idea of DTI is that it can extract a measure of directionality of the white-matter tracts in a given voxel, giving you a picture of where white matter is pointing in the brain. This can be used to infer which areas are strongly connected to each other and which less so. And, of course, many older techniques for measuring connectivity in animals - staining, tracing, etc. - are still widely used.

The other big exception to the rule of measuring correlation is techniques that can directly measure causality by disrupting some part of the system. If you can knock out part of the system and cause a part hypothesized to depend on it to fail, while knocking out the latter part doesn't affect the former, you can start to make some inferences about directionality of influence. In living humans, the latest way to do this is with transcranial magnetic stimulation (TMS), which seems to offer some ways to disrupt selected cortical areas temporarily, reversibly and on command. Although the technique is relatively new, it holds high promise as an additional tool in the connectivity toolbox. Other methods for disruption - cortical cooling, induced lesions in animals, even lesion case studies in humans - can provide valuable information on this front as well.

16.2.3 3. What are the different methods to analyze connectivity in fMRI? How do they differ from each other?

In a field burdened with a heavy load of meaningless acronyms and technical jargon, connectivity analyses stand out as a particular offender. There are what seems like a dizzying array of ways to model connectivity in fMRI, each with various acronyms and fancy-sounding concepts and a great number of equations underlying it. The important thing to remember in all of them is that the data input is essentially the same: it's just timeseries data. And the underlying computations are all essentially doing the same thing - looking for patterns in the data that are similar between regions. Some methods literally attempt to do the same thing, but for the most part, the different methods proliferate because they examine slightly different aspects of connectivity. So the important things to think about when faced with interpreting or performing any connectivity analyses are goals: what is the point of this analysis? What does its output measure? What are the alternate possible explanations that this analysis has ruled out, or failed to rule out?

There's a broad distinction you can make in connectivity analyses between model-driven and non-model-driven analyses. Non-model-driven analyses are those which don't "know" anything about the details of your experiment - some of them are called "blind" algorithms because they're searching for patterns in your data without knowing what the structure of your experiment was. These types of analyses can be used to get at activation in general, but they're probably more widely used in connectivity analyses. Principal components analysis (PCA) and independent component analysis (ICA) are non-model-driven types of analyses. I won't talk much about them yet here, because I don't know much about them right now. Anyone else out there, feel free to contribute some info...

The popular forms of model-driven analysis are those embedded into the popular neuroimaging programs, and SPM2 has recently added a couple to the field which are getting wide use. Psychophysiological interactions (PPIs) have been used in SPM and other programs for a while, but SPM2 has automated this analysis to make it a lot easier to perform. They start with a seed ROI and look for other regions that have high changes in connection strength to the seed as the experiment proceeds. Dynamic causal modeling (DCM) is Karl Friston's latest addition to the modeling tradition, and it's a much more all-encompassing form of connectivity analysis; he claims it subsumes all earlier forms of analysis as well as the standard general linear model activation analysis. DCM starts with a set of ROIs and attempts to determine the influence of each on the other and the experiment's influence on the connectino strengths. BrainVoyager is soon to release a connectivity package based on the concept of autoregressive modeling and Granger causality - a way of ruling out some directions of causality. This isn't out yet, due to a patent dispute, so I don't know much about it. Structural equation modeling (SEM) is used when you have a set of ROIs you'd like to investigate, but aren't sure what the links between them may be; it's a way of searching among the possible graphs that connect your ROIs and ruling out some connections while including others. Which of these you decide to use will decide on your experimental goals and what you want this analysis to show, exactly.

16.2.4 4. What is a psychophysiological interaction (PPI) analysis? How do I do it? Why would I want to?

A PPI analysis starts with an ROI and a design matrix. It's a way of searching among all other voxels in the brain (outside the seed ROI) for regions that are highly connected to that seed. One of the most straightforward ways of doing connectivity analyses would be to start with one ROI and simply measure the correlation of all other voxels in the brain to that voxel's timeseries, looking for high correlation values. As Friston and other pointed out a while ago, though, it's not quite as interesting if the correlation between two regions is totally static across the experiment - or if it's driven by the fact that they're both totally non-active during rest conditions, say. What might be more interesting is if the connection strength between a voxel and your seed ROI varied with the experiment - i.e., there was a much tighter connection during condition A between these regions than there was during condition B. That may tell you something about how connectivity influences your actual task (and vice versa).

PPIs are relatively simple to perform; you extract the timeseries from a seed voxel or ROI and convolve it with a vector representing a contrast in your design matrix (say, A vs. B). You then put this new PPI regressor into a general linear model analysis, along with the timeseries itself and the vector representing your contrast; you'll use those to soak up the variance from the main effects, which you'll ignore in favor of the PPI interaction term. When you estimate the parameters of this new GLM, the voxels where the PPI regressor has a very high parameter are those who showed a signficant change in connectivity with your experimental manipulation.

This is do-able in SPM99, or indeed any program; SPM2 makes it more automated, and adds some mathematical wrinkles, like deconvolving the HRF from your PPI regressor so as to look for interactions at the deconvolved (and hopefully neural) level, rather than at the HRF level.

PPIs are good to do if you have one ROI of interest and want to see what's connected with it. They're tricky to interpret, and they can take a really long time to re-estimate if you have several ROIs to explore and many subjects.

16.2.5 5. What is structural equation modeling (SEM)? How do I do it? Why would I want to?

Structural equation modeling analyses begin with a set of ROIs and nothing else. The idea in SEM is to try and estimate connection strengths between those ROIs that make up the best possible model of connection between them. The connection strengths are correlational (not directional), but represent the straightforward degree of correlation between the timeseries of those regions. This strategy (and variants of it) also fall under the title "path analysis," although that's a broader term that can describe analyses of non-timeseries data. SEM procedures can vary, but they're all kind of like the GLM: they search through the space of possible connection strengths until they find the set of connection strengths that best fits the data.

The measure of "best fit" is an important choice in SEM, and there's not wide agreement on the measure you should use, except a common suggestion that you use more than one and combine their results. Other bells and whistles on SEM analyses can include bootstrapping the data (see *P threshold* for information on permutation tests and bootstrapping) to get a confidence interval on how good the model could possibly be (Bullmore et. al (2000), NeuroImage 11, describe this strategy).

SEM isn't built in to any of the major neuroimaging programs that I know of, but several statistics program support it (as it's used in other social sciences besides neuroscience).

SEM is good to do when you have a set of ROIs - either functional or anatomical - and you're interested in knowing how strong the connections are between them (or whether connections between a particular pair exist at all) across the whole experiment (or part of it). It's a pretty straightforward style of analysis, but because of that, it doesn't take into account of lot of details of fMRI - temporal variations in the connection strengths, for example.

16.2.6 6. What is Granger causality? How does it relate to brain connectivity?

Granger causality is a concept imported from economics, where it was developed to do timeseries modeling of economic data (weird that that's the kind of data economists would want to look at - economic data, you know. Strange guys, those economists). It's an attempt to impose some directionality on connections between timeseries, or at least rule out some directions, by leveraging the rule of temporal precedence. The core of the Granger causality idea is that events can't cause events that already happened - so if a particular pattern happens in one timeseries, and then happens later in another timeseries, the latter one can't have caused the former one. Granger causation is a very limited form of causality, because it doesn't rule out the possibility that some third factor has induced the change in both of the timeseries, or any of the other problems common to ascribing causality to correlation data, but it's a start in the direction of blocking off certain directions of arrows.

The most explicit use of Granger causality has been in the connectivity package being developed for BrainVoyager, detailed below in Goebel et. al. The package is based also on the use of vector autoregressive modeling, which I couldn't begin to explain in detail but I gather is kind of like dynamic causal modeling or something like that. Unfortunately, the package has been held up in patent disputes, so it's not clear when we'll get to evaluate it up front. I'm not aware of other packages or programs currently using those methods to evaluate connectivity.

16.2.7 7. What is Dynamic Causal Modeling (DCM)? How do I do it? Why would I want to?

Well, this is another one that I'm undoubtedly going to botch the explanation for. But I'll take a very limited stab at it. If you're interested in this analysis, I highly recommend reading Friston et. al's paper on it at Connectivity.

DCM analyses are highly model-driven. You start with a set of ROIs and a guess at how they're connected with each other. That guess can be "fully connected," with every ROI attached to every other, or you can eliminate some connections off the bat. DCM then takes as its input your design matrix and the timeseries from those regions, and attempts a sort of hyper-advanced general linear model estimation. Instead of a general linear model, though, DCM explicitly considers some nonlinear aspects to the experiment: specifically, the connections between your ROIs and how they might change with the experimental manipulation. It goes through a huge set of Bayesian estimations and deconvolutions and every other fancy thing you can think of, and what you get on the way out is a big set of parameters. That set will include: HRFs for each of your regions, "resting" connection strengths between each of your regions, beta weights describing how the experiment affected each of your regions (just like regular beta weights), and "connection beta weights," indicating how the experimental manipulation affected your connection strengths. It'll also spit out some estimation of the statistical significance of each of these.

Friston et. al are hyped on this analysis; they believe that all the other analyses out there (SEM, PPI, etc.) are all special cases of DCM. Even the standard general linear model analysis of activation is a special case, they say, where you're assuming there are no connections between ROIs, and your ROIs are your voxels. A few papers have been put out thus far - Mechelli et. al (below) is one - using DCM in big analyses, with fairly promising results.

DCM is built into SPM2, and requires you to have SPM2 results to use it. It's not available yet for any other neuroimaging program.

DCM is great if you've got a set of ROIs, a hypothesis about how they might work, and you're particularly interested in how some areas or conditions might influence the connections between some other areas. Mechelli et. al is a good example of this - they looked at whether differences in visual activations due to categories of stimuli were mediated from the bottom up or from the top down. It's also kind of insanely complicated right now, and clearly in a sort of feeling-out phase in the community. Results may be difficult to interpret. But it's definitely the cutting edge of fMRI connectivity research for model-driven analyses right now.

16.2.8 8. How do I measure connectivity across a group?

Almost all of these methods measure correlations between timeseries, and so they're only appropriate to do at the individual level. The best way to run a group analysis is in the standard hierarchical fashion - take the output of the individual analysis and toss it into a group analysis. The output from all of them won't be the same - for PPIs, for example, you'll get an activation image, which works in SPM for a standard group-level analysis, whereas for SEM you'll get a set of connection weights, which you can then run a standard statistical test on in SPSS - but the hierarchical approach should work fine in general.

16.3 Contrasts

16.3.1 1. What's the difference between a T- and an F-contrast? When should I use each one?

Simply put, a T-contrast tests a single linear constraint on your model - something like "The effect size (parameter weight) for condition A is greater than that for condition B." T-contrasts can involve more than two parameters, but they can only ever test a single sort of proposition. So a T-contrast can test "The sum of parameters A and B is greater than that for parameters C and D," but not any sort of AND-ing or OR-ing of propositions.

An F-contrast, by contrast (ha!), is used to test whether any of several linear constraints is true. An F-contrast can be thought of as an OR statement containing several T-contrasts, such that if any of the T-contrasts that make it up are true, the F-contrast is true. So you could specify an F-contrast like "parameter A is different than B; parameter C is different than D; parameter E is different than F," and if any of those linear contrasts were significant, the F-contrast would be significant. The utility of the F-contrast is highest when you're just trying to detect areas with any sort of activation, and you don't have a clear idea as to the shape of the response. They were designed to be used with something like a Fourier basis set model, where you want to know if any combination of your cosine basis functions is significantly correlated with the brain activation. Testing that set with a T-contrast wouldn't be correct; it would tell you whether the sum of those basis functions' parameters was significant, which isn't what you'd want. Testing individually whether any of those parameters is significant, though, tells you something.

The disadvantage of the F-test is that it doesn't tell you anything about which parameters are driving the effect - that is, which of the linear constraints might be individually significant. It also doesn't tell you what the direction of the effect; parameter A might be different than parameter B, but you don't know which one is greater. This isn't a problem if you're using a basis set where different parameters don't have much individual physiological meaning (such as a Fourier set), but oftentimes F-tests are followed up with t-tests to further isolate which parameters are driving the effect and what direction the effect is in.

The Ward, Veltman & Hutton, and Friston papers on Contrasts both describe the F-test and how it's used in pretty clear fashion, with specific examples.

16.3.2 2. What's a conjunction analysis? How do I do one?

An F-test allows you to OR together several linear constraints, but what if you want to AND them together? That is, what if you want to test if all of a set of several linear constraints are satisfied? For that, you need a conjunction analysis. There are several ways to perform them - see the Price & Friston paper on Contrasts and those below it - but SPM provides a built-in way that is a good example. (Details of how to use SPM to do one are in the Veltman & Hutton paper there). The idea is to find the intersection of all the sets of voxels that satisfy a given linear constraint in the set, a simple mathematical operation in itself. The tricky part is to figure out what threshold level to use on each individual linear constraint to give the conjunction (or intersection) an appropriate p-threshold. SPM makes the choice that the p-thresholds on each individual constraint simply multiply together, so a conjunction of two constraints that you wanted to threshold at 0.001 would mean thresholding each individual constraint at the square root of 0.001. The resulting field of t-statistics is called a "minimum T-field" - effectively you're thresholding the smallest T-statistic among the linear constraints at each voxel - and SPM allows corrected p-thresholds to applied as well as uncorrected. These analyses are also available for F-constrasts, to AND together several OR statements.

One problem that some critics of this approach have highlighted is that it means at a voxel called "active" in the conjunction, any individual constraint on it may hardly be significant at all. If you want to see the conjunction of contrasts A and B, you'd prefer not to see 'common activations' that have p-values far above a reasonable threshold when looked at in each individual contrast. Price & Friston have argued that the individual constraints don't matter much in conjunctions, but some people still prefer not to use the minimum T-field approach for this reason. In this case, you can conjoin constraints together simply by intersecting their thresholded statistic maps (with some care taken to make sure the contrasts are orthogonalized (see below)), which can be done algebraically.

16.3.3 3. What does 'orthogonalizing' my contrast mean?

If you're testing a conjunction, one worry you might have is the the contrasts that make it up don't have independent distributions - that they are testing, to some degree, the same effect - and thus the calculation of how significant the conjunction of will be biased. If you use SPM to make a conjunction analysis through the contrast manager, it will attempt to avoid this problem by orthogonalizing your contrasts - essentially, rendering them independent of one another. The computation involved is complicated - not just simply checking whether the contrast vectors are linearly independent, although it's derived from that - but it can be thought of as follows:

Starting with the second contrast, check it against the first for independence; if the two are not orthogonal, remove all the effects of the first one from the second, creating a new, fully orthogonal contrast. Then check the third one against the second and the first, the fourth against the first three, and so on. SPM thus successively orthogonalizes the contrasts such that the conjunction is tested for correctly. See the help docs for spm_getSPM.m for more details.

16.3.4 4. How do I do a multisubject conjunction analysis?

Friston et. al is a good paper to check out for this. They describe some ways of thinking about the SPM style of conjunction analysis, which is normally a fixed-effects and hence only single-subject analysis, that allow its extension to a population-level inference. It's not clear that all the assumptions in that paper are true, and so it's on a little shaky ground.

However, it's certainly possible at an algebraic level to intersect thresholded t-maps from several subjects, just as easily as it is from several constraints. So it may make sense to try the simple intersection method, using somewhat loosened thresholds on the individual level. I'm not super sure on all the math behind this, so you might want to talk to Sue Gabrieli about this sort of thing...

16.3.5 5. What does the 'effects of interest' contrast image in SPM tell you?

Not an awful lot of interest, as it turns out. It's an image automatically created as the first contrast in an SPM analysis, and it consists of a giant F-contrast that tests to see whether any parameter corresponding to any condition is different from zero. In other words, if any of the columns of your design matrix (that aren't the block-effect columns) differ significantly from zero, either positively or negatively, at any voxel, that voxel will show up as significant in this F-image. Needless to say, it's not a very intepretable image for anyone who isn't using a very simple implicit-baseline design matrix. So generally, don't worry about it.

16.3.6 6. How is the intercept in the GLM represented in the analysis?

Every neuroimaging program accounts for the "whole-brain mean" somehow in its statistics, by which I mean whatever part of the signal that does not vary at all with time. That time-invariant point can be represented in the design matrix explicitly as a column of all ones, and SPM automatically includes a column like that for each session in a given design matrix. (AFNI and BrainVoyager don't explicitly show this column in the design matrix, but they include it in their model in the same fashion.) During the model estimation, a parameter is fit at each voxel to this whole-experiment mean, as any other column of the design matrix, and its value represents the mean signal value around which the signal oscillates. This is the 'intercept' of the analysis - the starting value from which experimental manipulations cause deviations. This number is automatically saved at each voxel in SPM (in the beta images corresponding to the block effect columns) and can be saved in AFNI or BrainVoyager if desired.

16.3.7 7. How do I make contrasts for a deconvolution analysis? What sort of contrasts should I report?

Generally, deconvolution analyses of the sort implemented by AFNI's 3dDeconvolve work on a finite impulse response (FIR) model, in which each peristimulus timepoint for each condition out to a threshold timepoint is represented by a separate column in the design matrix. In this case, a given 'condition' (or trial type) is represented in the matrix not by one column but by several. The readout of the parameter values across those peristimulus timepoints then gives you a nice peristimulus timecourse, but how do you evaluate that timecourse within the GLM statistical framework? There are a couple of ways; in general, the Ward is the best reference to describe them.

A couple obvious ones, though. First, an F-contrast containing a single constraint for each column of a given condition will test the 'omnibus' hypothesis for that condition - the hypothesis that there's some parameter significantly different from zero somewhere in the peristimulus timecourse, or more, simply, the hypothesis that there was some brain signal correlated to the task at some point following the task onset. This test won't tell you what sort of activity it was, but it will point out areas that had some sort of activity of some kind going on.

Secondly, a variety of different T-contrasts could be used to test various hypotheses about the timecourse. You might be interested in testing between two conditions at the same timepoint that you think might be the peak of the HRF. You might be interested in whether a single condition's HRF rose more sharply or fell more sharply (in which case a

T-contrast within that timecourse could be used). You might use some sort of a summing T-contrast to compare the 'area below the curve' in two different conditions.

There's not wide consensus about exactly what sorts of statistics count as 'significant' activation at this point in the literature - the difference between an HRF that rises sharply, spikes high, then falls back down to baseline quickly from an HRF that rises slowly, peaks only a little above baseline, but stays above baseline for a long time, isn't real clear at this point. No one is sure what such a difference represents exactly. This means, though, that there are a wealth of differences between timecourses that one could potentially explore. Almost any hypothesis can be made interesting with the right explanation, and fortunately almost any hypothesis can be tested in the GLM with the tools of T-tests, F-tests and conjunctions of constraints.

16.4 Coregistration

16.4.1 1. What is coregistration?

Remember realignment? It's just like that. It's a way of correcting for motion between images. But coregistration focuses on correcting for motion between your anatomical scans and your functional scans. The slightly trickier thing about that is that your anatomical scans might be T2-weighted, while your functionals are T1-weighted. Or maybe you have a Spoiled Grass anatomical and PET functional images. The intensity-based motion correction algorithms kind of choke on those. So coregistration aims for the same result as realignment - lining up two neuroimages - but uses different strategies to get there.

16.4.2 2. What are the different ways to coregister images?

These days, there are a few, but one de facto standard, which is coregistration by mutual information (MI). In some ways, coregistration is always the same problem - it's just like realignment, but can be between different modalities (PET, MRI, CAT, etc.), and usually you can be slower at it. The problem boils down to finding some function that measures the difference between your two images and then minimizing (or maximizing) it. Minimization/maximization is pretty standard these days; the question is what sort of cost function you use.

In realignment, we just used the sum of the squared difference in intensity between the images, measured voxel-by-voxel. The trouble with these scheme in coregistration is that your images might be different modalities, and hence a tissue type that's very dark in one (say, ventricle in PET) might be very bright in another (say, proton-density-weight MRI). In that case, trying to minimize the intensity differences between the images will give you a horrible registration.

So there are a couple strategies. SPM99 and older used templates within each modality that were already coregistered by hand with each other; that way, you could just realign your images to their modality-specific templates and automatically put them in register. These days, though, almost all automated coregistration schemes (including SPM2) use MI or some derivative of it.

16.4.3 3. What is mutual information, exactly?

In a nutshell: If two variables A and B are completely independent, their joint probability Pab (the probability that A comes up a at the same time that B comes up b) is just the product of their respective separate probabilities Pa and Pb; so Pab=Pa*Pb, if A and B are independent. On the other hand, if A and B are completely dependent - that is, knowing what value A takes tells you exactly what value B will have - then their joint probability is exactly the same as their respective separate probabilities; so Pab = Pa = Pb, if A and B are completely dependent. If A and B are dependent a little bit, but not entirely, their joint probability will be somewhere in between there - knowing what value A has tells you a little bit about B, so you can make a good informed guess at what B will be, but not know it exactly.

Mutual information is a way of measuring to what extent A and B are dependent on each other. Essentially, if you can estimate the true joint probability Pab for all a and b, and you know the individual probability distributions Pa and Pb, you can measure how far away the probability distribution Pab is from Pa*Pb, with a Kullback-Leibler statistic that measures the distance between curves. If Pab is much different from Pa*Pb, then you know A and B are dependent to some degree.

Alternatively, you can frame MI in terms of uncertainty; MI is the reduction in uncertainty about B you get by looking at A. If you're much more certain about A after looking at B, then A and B have high MI; they're quite dependent. If you don't know anything more about A after looking at B, then they have low MI and are pretty independent.

16.4.4 4. So how does mutual information help coregistration?

MI coregistration methods work by considering the intensity in one image to be A and the intensity in the other image to be B. The algorithm computes the MI between those two variables - finds the mutual information between the intensity in one image and the intensity in the other - and then attempts to maximize it.

The idea is that, instead of squared-intensity-difference methods which assume that a bright voxel in one image must be bright in another, you let the images themselves tell you how they're related. If by looking at a bright voxel in one image, though, tells you almost infallibly that the corresponding voxel in the other image is dark, then the images have very high MI, and they're probably close to registered. You can leave unspecified the relationship between intensities in the two modalities, and let the algorithm figure out how they're related - it automatically maximizes whatever relationship they have. This makes MI ideal for coregistering a wide variety of medical images.

16.4.5 5. How are coregistration and segmentation related?

Fischl et. al make the point that the two processes operate on different sides of the same coin - each one can solve the other. With a perfect coregistration algorithm, you could be maximally confident that you could line up a huge number of brains and create a perfect probability atlas - allowing you the best possible prior probabilities with which to do your segmentation. In order to do a good segmentation, then, you need a good coregistration. But if you had a perfect segmentation, you could vastly improve your coregistration algorithm, because you could coregister each tissue type separately and greatly improve the sharpness of the edges of your image, which increases mutual information.

Fortunately, MI thus far appears to do a pretty good job with coregistration even in unsegmented images, breaking us out of a chicken-and-egg loop. But future research on each of these processes will probably include, to a greater and greater extent, the other process as well. Check out *Segmentation* for more info on segmentation.

16.5 Design

The major tradeoff in planning your experimental design, from a statistical standpoint, is the fundamental one between efficiency and power. In the context of fMRI, power is the conventional statistical concept of how likely it is that your experiment will correctly reject the null hypothesis of no activation; it might be thought of at how good your experiment is at detecting any sort of activation at all. Efficiency, by contrast, is the ability to accurately estimate the shape of the hemodynamic response to stimuli - the variability that your average detectable hemodynamic response has. This is clearly important if you're interested in looking at response timecourse or HRF shape, but also important in finding activations at all - if variability in your modeled response is high, it's more difficult to distinguish one condition from another.

The tradeoff between power and efficiency is, unfortunately, inescapable (shown by Liu et. al) - you can't be optimal at both. Things that increase power include block design experiments, very high numbers of trials per condition, and increased numbers of subjects. Things that increase efficiency include designs with randomized inter-trial intervals (also called inter-stimulus intervals or ISIs) and analyzing your design with an event-related model (whether the design was blocked or not). Semi-random designs can give you a good dollop of both power and efficiency, at the cost of increased experimental length. Where you fall in designing your experiment will depend on what measures you're interested in looking at - but within the given constraints of a particular number of subjects, a reasonable experimental length, and a guess at how big an effect you'll have, there are good steps you can take to optimize your design.

Experimental design is heavily mixed in with setting your scanning parameters, and jittering your trial sequence, so be sure to check out the other design-related pages:

- *Scanning*
- *Jitter*
- *Physiology and fMRI*

16.5.1 1. What are some pros and cons of block designs?

Pros: High power, lower number of trials and subjects needed. Cons: Low efficiency, high predictability (which may be a problem for certain tasks from a psychological perspective).

16.5.2 2. What are some pros and cons of event-related designs?

Pros: High efficiency even at lower trial numbers, can have randomized stimulus types and ISIs. Cons: Low power, more trials/subjects needed, more difficult to design - efficiency advantages require jitter (see *Jitter*) or randomized ISIs.

16.5.3 3. What's the difference between long and rapid event-related designs? What's good and bad about each?

Long event-related designs have long ISIs - usually long enough to allow the theoretical HRF to return to baseline (i.e., 20 or 30 sec). Rapid event-related designs have short ISIs, on the order of a few seconds. Long event-related designs have generally fallen out of favor in the last few years, as proper randomization of ISI allows rapid designs to have much greater efficiency and greater power than long. Until the very late 1990s, it wasn't entirely clear that rapid event-related designs would work from a physiological perspective - that the HRFs for different trials would add roughly linearly. Since those assumptions have been (more or less) vetted, the only advantage offered by long event-related designs is that they're much more straightforward to analyze, and that rarely outweighs the tremendous advantages in efficiency and power offered by the increased trial numbers of rapid designs.

16.5.4 4. What purpose would a mixed (block and event-related) design serve? Under what circumstances would I want to use one? How do I best design it?

Mixed designs, which can include both block and event-related periods, or semi-random designs which have blocks of relatively higher and lower odds of getting a particular trial type, can give you good power and efficiency, but at the cost of longer experiments (i.e., more trials). They're more predictable than fully randomized experiments, which may be a problem for certain tasks. AFNI, SPM and Tom Liu's toolbox all have good utilities to design semi-random stimulus trains.

16.5.5 5. How long should a block be?

From a purely theoretical standpoint, as described by Liu and others, blocks should be as long as possible in order to maximize power. The power advantage of a block comes from summing the HRFs into as large a response as possible, and so the highest-power experiment would be a one-block design - all the trials of condition in a row, followed by all the trials of the next condition. The noise profile of fMRI, however, means that such designs are terribly impractical - at least one and probably two alternations are needed to effectively differentiate noise like low-frequency drifts from the signal of your response. So from a theoretical standpoint, Liu recommends a two- or three-block design (with two conditions, two blocks: on/off/on/off, with three conditions, two blocks: A/B/C/A/B/C, etc.). With few conditions, this can mean blocks can be quite long.

In practice, real fMRI noise means that two or three-block designs may have blocks that are too long to be optimal. Skudlarksi et. al, using real fMRI noise and simulated signal, recommend about 18 seconds for complex cognitive tasks where the response time (and time of initial hemodynamic response onset) is somewhat uncertain (on the order of a couple seconds). For simple sensory or motor tasks with less uncertainty in that response, shorter blocks (12 seconds or so) may be appropriate. Of course, you should always take into account the psychological load of your blocks; with especially long blocks, the qualitative experience may change due to fatigue or other factors, which would influence your results.

Bledowski et al. (2006), using empirically derived estimates of the HRF, mathematically derive a 7-sec-on, 7-sec-off block pattern as being optimal for maximizing BOLD response, suggesting it's a bit like a "swing" - pushing for the first half, then letting go, maximizes your amplitude.

16.5.6 6. How many trials should one block have?

As many as you can fit in to that time. The more trials the better.

16.5.7 7. How many trials per condition are enough?

In terms of power, you can't have too many (probably). The power benefits of increasing number of trials per condition continue increasing until at least 100 or 150 trials per condition (see Desmond & Glover and Huettel & McCarthy). In terms of efficiency, 25 or more is probably enough to get a good estimate of your HRF shape.

16.5.8 8. How can I estimate the power of my study before I run it?

Several of the papers below have detailed mathematical models for trying to figure that sort of thing out; if you can make an educated guess at how large (in % signal change) your effect might be from the literature, Desmond & Glover can give you a decent range of estimation.

16.5.9 9. What's the deal with jitter? What does it mean? Should I be doing it?

Jitter probably deserves its own FAQ, so check out *Jitter* for more info about it...

16.5.10 10. Do I have to have the same number of trials in all my conditions?

This question comes up especially for subsequent memory analyses, or things like it, where subjects might have only remembered a fraction of the trials they've looked at, but have forgotten a whole lot. If you're trying to compare remembered against forgotten in that case, is that okay? Depends on exactly the ratio. First and foremost, if a given condition has too few trials in general, you'll lose a lot of ability to detect activation in it - as above, if you don't have at least 25 trials in a condition in an event-related study (over the whole experiment), you're probably starting to get on thin ice in terms of drawing inferences between conditions. But the ratio of trial numbers between conditions can also have an influence. Generally, neuroimaging programs assume that the different columns of the design matrix you're comparing have equal variance, and a vast difference in numbers between them will invalidate that assumption. In practice, this probably isn't a huge concern until you're dealing with ratios of 5 or 10 to 1. If you have 35 trials in one condition and 100 in another - it's not ideal, but you probably won't be too fouled up. If you have 30 in one and 300 in another... it's probably cause for some concern.

16.5.11 11. How many subjects should I run? How many do I need?

Short answer: 20-25 subjects is a good rule of thumb. Long answer: Obviously this is affected to some degree by situations like funding, etc. But from a statistical perspective, this question boils down to what the levels of noise in fMRI are, or a power analysis: how many subjects should you have in order to detect a reasonably-sized effect a reasonable amount of the time? Using moderate estimates of effect sizes (0.5%) and estimating within- and between-subject noise from real data, Desmond & Glover (2002) calculated that 20-25 subjects were needed for 80% power (i.e., chance of detecting a real effect) with about the loosest reasonable whole-brain p-threshold. Smaller effect sizes might require more subjects for the same power, and looser p-thresholds (i.e., for an a priori anatomical hypothesis) might require fewer subjects. But in general, the 20-25 subject barrier is a pretty good rule of thumb. You aren't ever hurt by more subjects than that (although very large sample sizes can start tongues wagging about how small your effect size is, and you don't want to get into a fight about size - we're adults, after all). But unless you're very sure your effect size is much bigger than average, having fewer than 20-25 subjects means you're likely to be missing real effects. Check out Desmond & Glover for detailed analysis.

16.6 HRF

16.6.1 1. What is the 'canonical' HRF?

The very simplest design matrix for a given experiment would simply represent the presence of a given condition with 1's and its absence with 0's in a that condition's column. That matrix would model a signal that was instantly present at its peak level at the onset of a condition and instantly offset back to baseline with the offset of a trial. It's possible this may be a good model of neuronal activity, but since fMRI measures BOLD signal rather than neuronal activity directly, it's clearly not that good a model of the hemodynamic response that BOLD signal represents.

If we assume the hemodynamic response system is linear, linear systems theory tells us that if we can figure out the hemodynamic response to an instantaneous impulse stimulus, we can treat our real paradigm as the conglomeration of many instantaneous stimuli of various kind and the hemodynamic response should sum linearly. Tests over the last ten years suggest that the brain does, in fact, largely behave this way, so long as stimuli are spaced more than a couple hundred milliseconds apart. So the canonical HRF is a mathematical model of that impulse response function. It's a function that describes what the BOLD signal would theoretically be in response to an instantaneous impulse. Once your design matrix is described, your analysis software convolves it with a canonical HRF, so that your matrix now represents a gradual rise in activity and gradual offset that lines up with a 'typical' HRF.

Most of the common neuroimaging programs use similar canonical HRFs - a mixture of gamma functions, originally described by Boynton's group. This function has been found to be a roughly good model of hemodynamic response - at least in visual cortex - in most subjects. It models a gradual rise to peak (about 6 seconds), long return to baseline (another 10 seconds or so) and slight undershoot (around 10-15 seconds), the whole thing lasting around 30 seconds or so.

16.6.2 2. When should you use the canonical in your model? When should you use different response functions? (HRF, HRF w/ derivatives, etc.) What's the difference?

Generally, the canonical HRF is a decent fit to the true HRF for many normal subjects in many cortical and subcortical regions. If your analysis is intended primarily to test a hypothesis about neural activity, looking only for size and place of activation, and you believe your subjects to be reasonably normal, the canonical makes good sense. If you're looking to find out more about your activation than simply where it is and how big it is, though, you'll need to get a little fancier. Using the canonical HRF will tell you how much the canonical HRF (convolved with your design matrix) needed to be scaled to account for your signal. But you might be interested in more detail - how much variance there was in the onset of the HRF, or how much in its length, or the true shape of the HRF for your subjects. Or you might not think the canonical HRF is a good enough fit to your subjects or your region of interest for you (and there's certainly evidence to make that thought reasonable).

In those cases, you may want to complicate your model a bit. In the extreme, you could not use any sort of a guess at an HRF, and instead directly estimate the shape of your HRF by separately estimating parameters for every timepoint following your stimulus - a finite impulse response (FIR), or deconvolution model. We'll talk about those in more detail later in the course. Alternatively, you might choose to model your neural activity as a linear combination of basis functions like sines and cosines - this will guarantee you can get an excellent fit to your true HRF and use a different HRF at every voxel, thus avoiding the problem of regionally-different HRFs. This is a Fourier basis set model. As an intermediate step between the FIR/basis set - type models and the pure canonical HRF models, you might try modeling your activation as a combination of two or three functions - say, the canonical HRF and its temporal derivative, or dispersion derivative. This will separately estimate the contribution of the canonical HRF and how much variability it has in time of onset (temporal deriv.) or shape (dispersion deriv.).

There are tradeoffs for using the more complicated models: for FIR and Fourier models, the interpretation of any given parameter value becomes very different, and it becomes much more difficult to design contrasts. Using the intermediate steps can get you back physical interpretability, but at the price of decreased degrees of freedom in your data without a guarantee of better fit overall - and, in fact, Della-Maggiore et. al find that the HRF w/ temporal derivative has significantly decreased power relative to the canonical alone in a typical experimental design. So use at your own risk...

16.6.3 3. When does it make sense to do a regionally-specific HRF scan?

If you're particularly interested in a region that's not primary visual cortex and you'd like to get a very good fit of your model to the data, it may make sense to try and get an HRF that's specific to a different region. The canonical HRF is derived from measurements in V1 of some subjects, and studies like Miezin et. al have demonstrated that between-region variability within a subject in one scan can be significant.

However, if you're worried about this, rather than taking an entirely separate scan or task to estimate impulse response in a region, it probably makes more sense to use an analysis type that models the response of each voxel separately - like an FIR or Fourier basis set model - which doesn't assume that you have the same HRF at every voxel.

16.6.4 4. When does it make sense to do a subject-specific HRF scan?

If you're looking to study intersubject variability, or if you're looking to improve the fit of your model by a good chunk and you can afford the extra time in the scanner. Several studies, like Aguirre et. al and Miezin et. al, have demonstrated there is a significant amount of variability between subjects in several parameters of the HRF - time to onset, time to peak, amplitude, etc.

Perhaps more importantly, the canonical HRF is based on measurements from normal, adult cortex. It's becoming clear that populations like children, the elderly, or patients of various kinds may have HRFs that differ significantly from the canonical. Particularly in any sort of between-group study in which these populations are being compared to normal subjects, it's crucial to ensure that any effect you see isn't driven by the difference in fit of the HRF between groups - you would expect the canonical HRF to provide a better fit (and hence more and larger activations) to normal adults than it would to patients or non-standard populations. In cases like these, using a subject-specific HRF may be necessary or at least desirable.

16.6.5 5. Which regions have particularly different HRFs?

Probably a lot of 'em. But in particular there are questions about the extent to which the canonical HRF or others measured from cortical neurons maps onto HRFs for subcortical structures like the basal ganglia. Definitive answers about this sort of thing await further study. Logothetis & Wandell (2004) discuss some reasons why regions might differ in HRF - from increased white matter density to differences in vascular density to, um, complicated physiological things. But they have one clear point: we know HRF can vary from region to region, and there is no accurate way currently to convert the absolute BOLD magnitude to any neural measure. Which means comparing absolute BOLD effect between regions, even nearby, is simply not justified theoretically. In their words:

"It seems that with our current knowledge there is no secure way to determine a quantitative relationship between a hemodynamic response amplitude and its underlying neural activity in terms of either number of spikes per unit time per BOLD increase or amount of perisynaptic activity." - Logothetis & Wandell (2004)

So be cautious about comparing absolute magnitude between regions...

16.6.6 6. Which populations have particularly different HRFs?

Surprising few studies have been published on this subject. Ongoing studies at Stanford (Moriah Thomason & Gary Glover) suggest children of at least a certain age have important differences in their HRF, and it's clear the same is true of elderly subjects. But at present, if you're interested in using any non-standard population of subjects, it's not a crazy idea to assume they have significant differences in their HRF from the standard canonical.

16.6.7 7. What's the difference between an 'epoch' HRF and an 'event' HRF?

If you're using AFNI/BrainVoyager/SPM2, there isn't any. So don't worry about it. But if you're using SPM99 or earlier, the story above about the canonical HRF is actually oversimplified. In SPM99 (and earlier), epoch-related and event-related studies had the same underlying design matrix form - the onset of a trial was marked with a 1, but the actual trial itself was all 0's. Event-related studies were simply modeled by convolving those with a canonical HRF, but epoch-related studies clearly needed to account for the length of the trial. So there was a separate, epoch-related, canonical HRF, that was also based off a mixture of gamma functions, but was specifically scaled to account for the length of the trial - so the HRF that was convolved with the design matrix was different for a 12-second epoch and a 30-second epoch. The epoch HRF generally looked like a wider, fatter canonical HRF, and represented a model of linearly summed HRFs over the course of the trial.

With a re-vamping in data structures, though, in SPM2, and further study, this difference was scrapped. Epochs are now modeled differently at the level of the origingal, pre-convolved design matrix, with 1's down the whole length of the trial, and events and epochs are convolved with the same canonical HRF. This seems provide an equally good or better fit to real data as well as simplifying many calculation aspects.

16.6.8 8. What relationship does the BOLD have to the underlying neuronal activity?

(This should probably be higher up the question list.) The BOLD signal is produced by an influx of oxygenated blood to a local area of neuronal activity, to compensate for increased energy usage. But neurons use energy for a lot of things, both pre- and post-synaptic: action potentials, increased membrane potentials, cleaning up neurotransmitter, putting out neurotransmitter, etc. In order to correctly interpret the BOLD in a given region, we need to know if it's caused by, say, increased firing (i.e., output) or increased post-synaptic activity (i.e., increased input) or some combination. What aspect of the underlying electrophysiology does the BOLD correlate with?

Logothetis & Wandell review a good deal of this work. Several electrophysiology studies have found tight correlations between BOLD and local field potential (LFP), a lower-frequency electrical measure summed over many neurons (but still with good spatial resolution). Similarly, fast ERP amplitude seems to vary linearly with BOLD (Arthurs & Boniface, 2003). Slow ERPS, which are believed to arise from _post_synaptic potentials, correlate with BOLD in parietal cortex (Schicke et al., 2006)). Some studies have also found linear relationships between spike rates and BOLD, and spiking activity likely also correlates with BOLD, although perhaps not as robustly (Logothetis et al., 2001).

All this goes to suggest that BOLD may originate less in actual neuronal spiking and more in low-frequency potentials or increased excitability of neurons: in other words, BOLD reflects input to an area more than output. Clearly, input and output are often correlated for neurons; excitatory input will increase spiking outputs. But they aren't always; if this hypothesized connection is true, it means an increased BOLD could reflect increased inhibitory input to a region, or summing of both inhibitory and excitatory inputs (resulting in no change in spiking). Logothetis & Wandell also cite examples where BOLD might be different from single-unit recording - e.g., an area may appear highly direction-sensitive with BOLD, even if single-unit recordings show it not to be, because it is highly interconnected with a very direction-sensitive area. Attention effects, which are difficult to find in V1 with single-unit recordings but show up in fMRI, might be another example. So we should have caution in using models that require BOLD signal to directly index increased spiking outputs.

This is not to say it's impossible to map single-unit firing onto BOLD signal; retinotopy in visual cortex, for example, happens at the single-unit level, and is easily detectable with BOLD. But merely note: many factors, from input activity to vascular density (see above), etc., can affect regional BOLD response.

16.6.9 9. How does the hemodynamic response change with stimulus length?

In very nonlinear ways. See Glover (1999) (and Logothetis & Wandell, 2004). You're on very shaky ground if you attempt to model stimuli longer than 6 sec by convolving a standard HRF with a boxcar. You may be better off using stimuli separated by longer times or shorter stimuli.

16.7 Jitter

Jittering is heavily mixed in with experimental design and setting your scanning parameters, so be sure to check out the other design-related pages: *Design*, *Scanning* and *Physiology and fMRI*

16.7.1 1. What is jittering?

It's the practice of varying the timing of your TR relative to your stimulus presentation. It's also often connected to, or even identified as, the practice of varying your inter-trial interval. The idea in both of these practices is the same. If your TR is 2 seconds, and your stimulus is always presented exactly at the beginning of a TR and always 10 seconds long, then you'll sample the same point in your subject's BOLD response many times - but you might miss points in between those sampling points. Those in-between points might be the peak of your HRF, or an inflection point, or simply another point that will help you characterize the shape of your HRF. If you made your TR 2.5 seconds, you'd automatically get to sample several other points in your response, at the expense of sampling each of them fewer times. That's "jittering" your TR. Alternatively, you might keep your TR at two seconds, and make the time between your 10-second trials (your inter-stimulus interval, or ISI) vary at random between 0 and 4 seconds. You'd accomplish the same effect - sampling many more points of your HRF than you would with a fixed ISI. You'd also get an added benefit - you'd "uncover" a whole chunk of your HRF (the chunk between 10 seconds and 14 seconds) that you wouldn't sample at all with a fixed ISI. That lower portion can help you better determine the shape of your whole HRF and find a good baseline from which to evaluate your peaks. This added benefit is why most people go the second route in trying to "jitter" their experiment - varying your ISI gets you all the benefits of an offset TR, plus more.

16.7.2 2. Why would I want to do this in my experiment?

If all you care about is the amplitude of your response, you probably wouldn't. In this case, you're assuming a certain shape to the hemodynamic response, and all you care about is how "high" the peak of the HRF was at each voxel for each condition. You'd want a design with very high statistical power - the ability to detect amplitude. On the other hand, you might not want to assume that every voxel had the same HRF shape for every condition. If you'd like to know more about the shape, without assuming anything (or less than everything, at least), you need a design with high statistical efficiency - the ability to accurately estimate shape parameters, without assuming a shape.

Varying your ISI is a strategy to increase the efficiency of your estimates at the expense of your power. Clearly, because you'll be sampling each point of your HRF fewer times, you'll necessarily have less confidence in the accuracy of any given estimate. But because you'll have so many more points to sample, you'll have much more confidence about the true shape of your HRF for that condition. This is critical when you believe your experiment may induce HRFs of different shape in different region, or if knowing the shape (lag, onset time, offset time, etc.) of your response is important (as it is in mental chronometry). With a variable-ISI design, you can run a rapid event-related design and pack many more trials into a given experimental time than you would for a fixed-ISI design that sampled the whole HRF, or you can sample much more of the HRF than a fixed-ISI design could with the same number of trials.

16.7.3 3. What are the pros and cons of jittering / variable-ISI experiments? When is it a good/bad idea?

Variable-ISI experiments are a way of making the tradeoff between power and efficiency in an experiment. Fixed-ISI designs are extremely limited in their potential efficiency. They can have high power by clustering the stimuli together: this is a standard block design. But in order to get decent efficiency in an experiment, you need to sample many points of the HRF, and that means variable ISI. Any experiment that needs high efficiency - say, a mental chronometry experiment, or one where you're explicitly looking for differences in HRF shape between regions - necessarily should be using a variable-ISI design. By contrast, if you're using a brand new paradigm and aren't even sure if you can get any activation at all with it, you're probably better off using a block design and a fixed ISI to maximize your detection power.

From a psychological standpoint as well, the big advantage of variable-ISI designs is that they seem far more "random" to subjects. With a fixed ISI, anticipation effects can become quite substantial in subjects just before a stimulus appears, as they catch on to the timing of the experiment. Variable ISIs can decrease this anticipation effect to a greater or lesser degree, depending on how variable they are.

16.7.4 4. How do I decide how much to jitter, or what my mean ISI should be?

Great question. Depends a lot on what your experimental paradigm is - how long your trials are, what psychological factors you'd like to control - as well as what type of effect you're looking for. Dale et al. lays out some fairly intelligible math for calculating the potential efficiency of your experiment. Probably even easier, though, is to use something like Tom Liu's experimental design toolbox. Once you're within the ballpark for the type of paradigm you like, these tools can be an invaluable way to optimize your design's jitter / ISI variation, and are highly recommended for use.

16.7.5 5. But how do I get better temporal resolution than my TR?

Simple: don't always sample the sample points of your response. If you always sample the BOLD response 2 seconds and 4 seconds and 6 seconds after your stimuli are presented, for your whole experiment, you'll have a very impoverished picture of the shape of your HRF. But if, for example, you sampled 2 sec. and 4 sec. and 6 sec. post-stimulus for half the experiment, then cut one second between trials and sampled 1 sec. and 3 sec. and 5 sec. for the rest of your experiment - why, then, you'd have a better picture. The cost, of course, is reduced power and expanded confidence intervals at the points you've sampled.

With a good picture of the shape of your HRF, though, you could then compare HRFs from two different regions and see which one had started first, or which one had reached its peak first. If HRF timing is connected in some reliable way to neuronal activation, you then don't need to sample the whole experiment at a super-fast rate - you could infer from only a limited-sample picture of one part of the HRF where neuronal activity had started first and where it had started second, which allows you to rule out certain flows of information.

16.8 MATLAB Basics

Here's a very quick intro to some of the basic concepts and syntax of MATLAB. The hope is that after covering some of these basics, you'll be able to troubleshoot SPM and MATLAB errors to some degree. Experienced MATLAB programmers may find most of this redundant, but there's always more you can pick up in MATLAB, so please, if you experience folk've got tips and tricks to suggest, add them onto these page!

This part looks specifically at the most basic of basics. For info on how to read m-files and m-file programming concepts, check out *MATLAB Programming*. As well, *MATLAB Paths* answers questions about modifying your search paths, and *MATLAB Debugging* touches specifically on strategies for troubleshooting - how to get more information, what parts of error messages to look for, and how you can dig through a script to find what's gone wrong.

If you take nothing else away from this page, take this: The MATLAB tutorial, contained within the program's help section, is terrific. It'll teach you a lot of the basics of MATLAB and a lot of things you wouldn't think to learn. The progam help is always a great place to go for more info or to learn something new. Nothing will help your troubleshooting and programming skills like curiousity - when you hit something you don't know or recognize, spend a couple minutes trying to figure out what it is, with the MATLAB and SPM help. If your help within the program isn't working, it's all online at the Mathworks website (http://www.mathworks.com/help/). As well, if you type help command into MATLAB, where command is some MATLAB function name, you'll almost always get a quick blurb telling you something about the function and how it's supposed to operate. It's invaluable for reference or learning on the fly.

16.8.1 MATLAB Language

Here are some quick basics on MATLAB code and what it all means. You don't need to be expert on all of this, but it may help you understand what an error message or piece of script means - a sort of Rosetta stone, if you will.

Variables

The very very first thing about MATLAB to understand is what a variable is. A variable in MATLAB is like a variable in algebra - it's a name given to some number. In MATLAB, you can name any single number (like a = 4) or multiple numbers at once, in which case the variable is a vector or an array of numbers (like a = [1 2 3 4]). A single number is called a scalar variable in MATLAB, but a single variable name can represent a huge array of numbers of arbitrary size and dimension. Creating a variable or manipulating it is done with = statements, where the left side of the = is your variable name, and the right side is the value you want it to take. If the name doesn't already exist, an = statement will create it - something like my_variable = 6005. If the name is already taken for a variable, an = statement can be used to re-assign that variable name to something else entirely, or simply manipulate the values within that variable. You can change a whole variable at once or simply a few of the numbers (or characters, or whatever) within it.

Workspace

When you start MATLAB, it opens a chunk of memory on the computer called the "workspace." When you create a variable, it exists in the workspace, and it stays there until you clear it, or until you re-assign that variable name to another value (which you can do freely). Only variables in the workspace can be used or manipulated. You can easily tell what variables are present in the workspace with the command whos . You can save a whole workspace to the disk with the command save filename; this will create a binary file with a .mat extension in your present working directory (more on the working directory later). That .mat file contains all the variables you've just saved, and you can clear the workspace and then load those variables again with the command load filename. You can clear the whole workspace at once with clear or clear any individual variable name with clear variable.

Array notation

Sets of numbers (or characters, or other things) in MATLAB are arranged in arrays - basically rectangles of rows and columns of numbers. Arrays can be only a single number - those are called scalars. They can also be single rows or columns - those are called vectors. Two-dimensional arrays (with multiple rows and columns) are pretty common, but arrays can also be three-dimensional, or even four- or more-dimensional. If you want to access a single element of the array, you use array notation, which puts parentheses after the variable name to specify which element. So if I have a 4-by-4 array called a, a(3,2) refers to the element in the third row, second column. Array notation always refers to row first, column second, other dimensions afterwards in order. You can type a(3,2) by itself in MATLAB and it'll tell you what's in that spot, or you can use = to reassign that number, with a command like a(3,2) = 16. If you have a vector, you can use only one number in the parentheses - something like a(4) = 56. You can make arrays by using brackets. a = [1 2 3 4] will make a four-element row vector - a 1x4 matrix. Separating the elements by spaces will put them in different columns on the same row. Separating them by semicolons will put them in different rows in the same column, so a = [1;2;3;4] will make a column vector, or a 4x1 matrix. You can combine the two methods. a = [1 2 3 4; 1 2 3 4; 1 2 3 4; 1 2 3 4] will make a 4x4 matrix, with 1 2 3 4 on each row. Using the colon vector to specify a range of numbers can be a quick way to build matrices this way. There are also several commands, like eye, ones, zeros, and others, that build specify types of matrices of a specific size automatically.

Variable types

There are several types of variables you can use in MATLAB. These include various kinds of numbers - integer, floating point, complex - as well as alphanumeric charactetrs (arrays of characters can be treated as strings). A given standard array can only hold one 'type' of thing - an integer array can only have integer values, a character array can only have characters as values, etc. Sometimes, though, you'd like to have more than one type of information in a given array, lumping several different-sized number arrays together, or lumping some numbers and strings and so forth together into one unit. For those, you can use cell arrays and structures, two specialized forms of variables.

Cell arrays

Cells array are like normal arrays, but each element of a cell array can contain anything - any number type, any other array, even more cells. Cell arrays are accessed like normal arrays, but using curly braces instead of standard parentheses, so if I have a cell array c, I might say c{3,2} = 16, followed by c{3,3} = 'cat'. I can refer to the cell itself with standard parentheses - so c(3,3) is a cell - or the contents with the curly braces - so c{3,3} is a character array above. I can refer to the cell's contents in an expression like this by following the curly braces with the standard array notation. If I have the above array, c{3,3}(2) would be a single character - 'a'. You can make cell arrays simply by using the curly braces: mycell{1,1} = 'hello' would make a single cell containing the string 'hello'. The command cell will also make empty cell arrays. There are also various utilities to convert arrays back and forth between cell arrays and standard arrays.

Structures

Structures are like regular variables that are made up of other variables. So a single structure contains several 'fields,' each of which is a variable in its own right, with a name and a particular type. Fields can be of any type - numbers, letters, cells, or even other structures. SPM makes enormous use of structures in its data organization, particularly structures nested within other structures. If you have a structure called s, it might have subfields called data and name, and you could access those with a dot, like so: s.name = 'Structure, or s.data = [1 2 3 4]. Nested structures are accessed the same way; you could access a sub-subfield by something like s.substructure.smallstruct.data = 3. You can make a structure with the struct command, where you specify fields and values, or you can create them simply by naming a new variable and using a dot - so if there was no variable called s, the command s.name = 'Structure' would create a structure called s with a single field called name. You can add fields just by naming them in the same way, and remove them with the rmfield command. Structures can be put in arrays or vectors, so long as all the structures in the array or vector have the exact same field names and types. So you could reference s(4).data if you had 4 structures all in a vector called s. Typing the name of a structure will always tell you all its fields and what they are.

The colon operator

Last among the real basic things in MATLAB is the colon operator, which operates as shorthand for a whole range of numbers at once. The expression 1:10 is shorthand for "every number between one and ten, inclusive, counting by ones". You can specify a different number to count by simply by putting a different number between two colons, so the expression 1:2:10 is every number between 1 and 10, counting by 2s (1, 3, 5, 7, 9). You can use the colon operator to make values for arrays: the command a = 1:10 makes the row vector [1 2 3 4 5 6 7 8 9 10]. You can also use it in indexing other arrays, where it's particularly powerful. If a is a 4x4 array [1 2 3 4; 1 2 3 4; 1 2 3 4; 1 2 3 4], the expression a(2:4, 2:4) gives me back a new matrix which is the second through fourth rows and columns: [2 3 4; 2 3

4; 2 3 4]. I can use the colon operator by itself as shorthand for "all rows" or "all columns;" with the 4x4 a array, the expression =a(:,1:2) would give me [1 2; 1 2; 1 2; 1 2].

Using all notation together

It'll take a while if you haven't programmed before, but eventually you'll learn to use all of these notations together, which make the language particularly powerful. An expression like Sess{3}.U(4).C.C(1:10, :) may look horrible on the face of it, but all it's doing is picking out a small submatrix; that submatrix is embedded in (reading right-to-left): a larger matrix (Sess{3}.U(4).C.C) inside a structure (Sess{3}.U(4).C) inside a vector of structures (Sess{3}.U) inside a cell (Sess{3}) which is part of a cell array (Sess). That submatrix would be the 1st through 10th rows of C and all of its columns. When in doubt on what an expression means, use whos and type the names of variables and fields to try and figure out what they are. Read backwards (from right to left) and try and work out the organization of the expression that way.

Miscellaneous stuff

The variable name ans is built into MATLAB as shorthand for "the result of the last thing you typed in." You can assign the results of commands to other variables, like b = a(1:3, 1:3), but if you don't and just type a(1:3, 1:3), the result of that command (a 3x3 matrix) will be given the name ans. The thing to be aware of is that the next time you type a command with unassigned output - say the next thing you typed was a(2:4,2:4) - the variable name ans would be reassigned to take on the value of the latest result. Whatever was in ans before is lost! So be careful assigning your output to variable names. And if you get something in ans that you want to keep, put it in another variable right away, by saying something like myvar = ans.

Generally, MATLAB will report the result of your command to the screen, telling you what the new value of your result variables was. If you don't want it to do that - because, for example, your result will be a 2000x2000 matrix - you can put a semicolon at the end of the line and MATLAB will suppress its output. So a = ones(2000,2000); will generate a 2000x2000 matrix of all ones, but it won't print that matrix to the screen. In m-files, you'll see almost every line ends with a semicolon so that you're not barraged during a function or script's execution by results whipping by.

16.8.2 MATLAB Debugging

This page looks specifically at how to troubleshoot MATLAB problems - how to get more information, how to read and interpret error messages, and how to dig through m-files to find what's gone wrong. For the most basic of basics, check out *MATLAB Basics*. As well, *MATLAB Paths* answers questions about modifying your search paths, and *MATLAB Programming* talks about how to read m-files and how to program with them.

Troubleshooting SPM and MATLAB

So there you are, tootling along estimating your model or coregistering your anatomy or what have you. And then, boom: computer beeps, error message pops up, everything stops, fire and brimstone come from the sky. What can you do about it? Lots, as it turns out. Armed with a sense of the MATLAB basics and understanding of MATLAB code, you can often divine a lot about what's going on in your error from your crash - often enough to fix the problem, but at least enough to narrow down the possibilities. The two crucial entry strategies to troubleshooting are: knowing where the problem happened, which frequently boils down to interpreting your error messages, and knowing what was happening when the problem hit, which relies on MATLAB's debugging package. We'll tackle them separately.

Non-errors

Before we jump into errors, a quick word about non-error problems. Warnings sometimes crop up in MATLAB, and they allow the program to continue. They're not generally anything worth worrying about - they'll let the program run - but they're worth noting, and trying to understand what they mean, particularly when they precede an error. The programmer puts warnings in for a reason, to notify you of something weird happening, so listen to her...

Also, people sometimes ask whether they can tell if an SPM program is crashed or just running for a long time. In general, MATLAB rarely crashes outright without generating some kind of error message, so if something's stopped responding and you've got a "busy" message but no error message, it's probably still working. If it seems like it's been going for an inordinate amount of time, though, check in with somebody.

Where did the error happen? or, Everything you need to know about error messages

SPM errors can happen in terribly uninformative fashion. Everyone who's used SPM has seen the infamous "Error during uicontrol callback," which is a generic MATLAB error that means "Something went wrong in the function called by pushing whatever button you just pushed." Fortunately, MATLAB has relatively well-designed error-detection and bug-tracking facilities. You can learn a great deal about your error just by interpreting your error message.

Sometimes in SPM you'll get a full error message of several lines, but oftentimes you'll get the one-line "uicontrol callback" error. However, if you close the panel that's home to the button that generated the error, you can often get the full underlying error message. So if you hit "volume" in the results section and get a uicontrol callback error, closing the results panel will usually generate further information about the error. You may also be able to generate more information simply by going to the MATLAB window and hitting return a couple of times. Always try and get a full error message when you have an error! Even if you can't figure out the problem, it's almost totally useless to another troubleshooter if you come to them and say, "I have an error!" or "I have a uicontrol callback error!" without any more information about the problem. Gather as much information about the error as you can - the MATLAB window is your friend.

MATLAB error messages are of varying helpfulness, but they all follow the same structure, so let's take a look at one:

```
??? Error using ==> spm_input_ui Input window cleared whilst waiting for response:␣
↪Bailing out!
Error in ==> /usr/local/MATLAB/R2014a/toolbox/spm12/spm_input.m
    On line 77 ==> [varargout{:}] = spm_input_ui(varargin{1:ib-1});
Error in ==> /usr/local/MATLAB/R2014a/toolbox/spm12/spm_get_ons.m
    On line 146 ==> Cname{i} = spm_input(str,3,'s',sprintf('trial %d',i),...
Error in ==> /usr/local/MATLAB/R2014a/toolbox/spm12/spm_fMRI_design.m
    On line 225 ==> [SF,Cname,Pv,Pname,DSstr] = ...
Error in ==> /usr/local/MATLAB/R2014a/toolbox/spm12/spm_fmri_spm_ui.m
    On line 289 ==> [xX,Sess] = spm_fMRI_design(nscan,RT);
??? Error while evaluating uicontrol Callback.
```

SPM errors will often look like this one: a "stack" of errors, although the stack may be bigger or smaller depending on the error. So what does it mean?

First, the order to read it in. The top of the stack - the first error listed - is the most immediate error. It's what caused the crash. MATLAB will tell you what function the error was in - in this case, it's spm_input_ui - and some information about what the error was. That error message may be written by the function's programmer (in which case it's often quite specific) or be built-in to MATLAB (in which case it's usually not). Sometimes that top error may also give you a line number on which the error was generated within the function.

The rest of the stack is there to tell you where the function that crashed was called from. In this case, the next one down the stack was in spm_input - so our error was in spm_input_ui, which was called by spm_input. The line in spm_input that calls spm_input_ui is highlighted for us - it's line 77 within spm_input.m. The next error down tells us that spm_input was itself called from within spm_get_ons, on line 146, and so on. The last error tells us that the very first function that was called was spm_fmri_spm_ui - that's the function that was called when we first hit a button.

From only this limited information, you can often tell a lot about the error. One important distinction to draw is whether the function where the error happened was an SPM function or a MATLAB function. Built-in MATLAB functions are generally pretty stable, and so if they crash, it's usually because you, the user, made a mistake somewhere along the line.

When the function that generated the error is an spm function, you can usually tell - it'll have the prefix *spm_* or generally sound like something from fMRI analysis and not MATLAB generally. In this case, the error is sometimes in the code and sometimes a mistake earlier in your analysis. The error messages can sometimes tell you enough to figure out what's going on: perhaps it says that there's an index out of range, and you realize it's because you're trying to omit the 200th scan from a session that only has 199. Just looking at the line that generated the error and what the actual error message says can give you a lot of info off the bat. Study the message carefully and look at the function stack it came from and see if you can figure out what the program was doing when it crashed.

It's not always enough to look at the problem after it's crashed, though. Sometimes you need to figure out what's happening when it actually crashed, and in that case, you need MATLAB's debugging package.

MATLAB Debugging

MATLAB's debugging package is pretty good. Hard-core debugging is a skill that's acquired only after a lot of programming, but even someone brand-new to reading code can use some debugging techniques to make sense of what's going on with their data, and hopefully learn some details about MATLAB code at the same time.

You'll need to be running the graphical version of MATLAB to take real advantage of MATLAB's debugging; without it the text editor won't open and you won't be able to see the details of the function you're debugging.

The starting point for any bug-finding expedition is to answer the question, "What was happening in the program when it crashed?" The hope is answering that will tell you why it crashed, and how to keep it from crashing again. When errors ordinarily occur in MATLAB, they end the execution of the program and close the workspace of the error-ing function, losing all its variable and so forth. The debugging package allows you, instead, to wait for an error and then "freeze" the program when one happens, holding it still so you can examine the state of all its variables and data.

If you want to try debugging an error, figure out when your error happens first (during model estimation or when you push "load" or whatever). Then type "dbstop if error" at the MATLAB prompt. This will engage the debugger in "waiting" mode. Then run your program again and do whatever caused the error. This time, you'll get an error message, but the prompt will change to K>>, and a text window will pop open containing the function that caused the error. The K prompt means you have Keyboard control - the program is frozen, and you can poke around as much as you like in it. To end the program when you're done debugging, type "dbquit" at the K>> prompt, and the program will end.

Once you have the program frozen, it's time to look around. Check out the line that generated the error and see if you can figure out what the error message means. If it's called by an error function, is it inside an if block? If so, what condition is the if block looking for? If the error is caused by an index out of range, can you figure out what the index variable is? What is its value? What is the array that it's indexing? What sort of information is in there? Type whos to figure out what variables are currently in the workspace and what types they are, and type the name of variables to find out their values.

It may not be immediately clear why a variable has the value it does or what it represents. Try reading line-by-line backwards through the function and find out when the variable got its current value assigned. Where did it come from? Are there any comments around it telling you what it is? Are there any comments at the top of the function describing it?

Sometimes a variable may have a strange value passed into it by whatever function called the error-ing function. The MATLAB debugger has frozen that function as well, and you can move 'up' into its workspace to take a look around in the same way. Type "dbup" at the K>> prompt, and you'll switch into the workspace of the function that called the current one. You can go up all the way to the base workspace, and "dbdown" will let you switch back down through the stack. Remember that each workspace is separate - their variables don't interact, at least when all the m-files are functions.

There are other debugging commands as well, more useful for you in writing your own scripts. Check out the MATLAB help for keyboard, dbcont, dbstep and things like that.

There's not more specific debugging advice to give, besides go explore the function. Every error is a little different, and it's impossible to tell what caused it until you get into the guts of the program. Good luck!

16.8.3 MATLAB Paths

This Part looks specifically at how to read and modify your search path. For the most basic of basics, check out *MATLAB Basics*. As well, *MATLAB Programming* talks about how to read m-files and program with them, and

MATLAB Debugging touches specifically on strategies for troubleshooting - how to get more information, what parts of error messages to look for, and how you can dig through a script to find what's gone wrong.

Paths

Every time you type a name in MATLAB - of a variable, a function, a script, anything - the program goes through a certain procedure to figure out what you're referring to. It'll first look in its workspace for variables of that name; if it doesn't find anything, it will assume you're trying to refer to a function, script, or .mat file. MATLAB always looks first in the present working directory for the .m file or .mat file of the given name, but if it doesn't find one there, it has a specified sequence of directories it looks in.

This directory list is called the path. It's similar to the Linux concept of the same name. Maintaining and manipulating your path can be crucial to running MATLAB programs and keeping track of different versions of functions. Fortunately, MATLAB makes it pretty easy to work with paths.

You can always look at your path in MATLAB by typing path; MATLAB searches the output list from top to bottom. The easiest way to work with your path is by typing "pathtool" at the MATLAB prompt. That'll open up a graphical interface that will display your current path, let you easily add or remove directories, or re-order what's in there already.

If you're curious where on your path a particular m-file is, you can type which filename; this will tell you the location of the first copy of that file MATLAB hits on the path, and hence which version of the file it will run. If you type open filename and filename is on your path, open will open the first copy it hits on the path. These two commands can be super helpful in figuring out whether you've got the right version of code, as well as figuring out where functions are located.

In general, any manipulations of the path you do are good only for your current session of MATLAB; when you shut it down and restart it, you'll go back to your default path. The default path in MATLAB is stored in a file called pathdef.m. But you can change your own default path! All you need is a startup.m file.

Startup.m files

When you launch MATLAB, it will automatically search for an m-file called startup.m in a subdirectory of your home directory. If it doesn't find a startup.m file there, no biggie - nothing happens. If a file called startup.m exists there, though, it will automatically run that file. This can be a super useful way for you to customize MATLAB for your own use. There are a number of commands you can run in MATLAB to change output or parameters, but one obvious thing to do is customize your path.

The command addpath('directoryname'); in a startup.m file will automatically ensure that directoryname is the first directory on your path whenever you start MATLAB. If you have several addpath statements in there, they'll run in order, adding to the top each time, so the last addpath statement specifies the first directory on your path. This is really helpful if you want to keep your own MATLAB code (or MATLAB data) available above anything else. Simply add the directory where you keep the code or data to the top of your path, and you'll always be able to run it, no matter where your present working directory is.

16.8.4 MATLAB Programming

This part looks specifically at how to read m-files and m-file programming - functions, scripts, etc.. For the most basic of basics, check out *MATLAB Basics*. As well, *MATLAB Paths* answers questions about modifying your search paths, and *MATLAB Debugging* touches specifically on strategies for troubleshooting - how to get more information, what parts of error messages to look for, and how you can dig through a script to find what's gone wrong.

Functions

Once you've figured out some basic ways to create and refer to your variables and store some information in them, you probably want to manipulate that information. That's where functions and scripts come in.

A function is essentially a shorthand name for a whole set of commands. They're stored in m-files - text files with .m extensions. Any file with a .m extension you can generally read with a standard text editor, and it'll just contain a whole set of MATLAB commands. Conventionally, you put only one command per line of the file, and when you run the function, MATLAB just reads through the file top to bottom, executing each command in order. You run a function simply by typing the name of its m-file, assuming that m-file is in the MATLAB path.

Functions often will take some data as input and spit back out some data as output. Functions are referred to kind of like arrays, but instead of supplying indices, you supply input variables. So saying spm_vol(VY) is calling the function spm_vol and passing it the input variable VY. If you said mydata = spm_vol(VY), you'd be saying you wanted whatever output that spm_vol chose to spit out to be stuck into the variable mydata.

Scripts

A .m file can contain either a function or a script. The difference between the two is subtle but important; it has to do with what workspace you have access to when you're executing the commands inside your m-file. You can easily tell the two apart; functions always have the function command as their first executable line, which will specify the name of the function and what its input and output are. Scripts don't have the function command - they just launch right in.

The difference between the two is in the workspace. When you run a function, it opens its own, clean workspace. Any variables you have in the workspace when you run the function can't be accessed inside the function, unless you pass them in as input variables. When it ends, it closes its workspace and any variables there that aren't designated as output variables are lost.

Scripts, by contrast, operate in the same workspace from which they're called. So any variable you have in the workspace when you call a script can be manipulated by the script, and any variable created in the script will be left in the workspace when the script ends.

Reading .m files

If you're running the graphical version of MATLAB, you can open the built-in text editor with the command open filename. The single best thing you can do to learn and teach yourself about the inner workings of SPM (or other MATLAB packages) is to open up particular functions and read through them to see what they're actually doing. Reading them can seem like reading Greek a lot of the time, but it's a skill you can learn, and it'll help you understand what's happening to your data better than anything else. Unless you can read MATLAB code, you'll always be a little in the dark about what's being done in your analysis. You don't need to know how to program the code - just to understand more or less what it's doing.

Reading code is relatively straightforward in some ways - you simply start from the top and go through line-by-line to see what each successive line does. But there are some things to keep in mind on the way:

- Lines that start with % are "comments." Anything after a % on a line is ignored by MATLAB (even if there was real code before the % on the line), and so comments are used by programmers to explain what's happening in a file. Most functions have some overview comments at the top of the file, to try and explain what's happening overall and hopefully some of the data structures and variables in use. Good programmers also comment throughout the function, to point out different sections of code and help explain what's happening in each chunk, or why they chose to use a particular strategy. Read the comments! They can be super helpful in understanding what's going on.
- If you see a function name and don't know what it does, check the MATLAB help! There's a lot of information in there about what every function under the sun does. Many MATLAB functions are obviously named, but

not all. If it's not a MATLAB function, try typing help function-name at the prompt anyways - a lot of times something will come up.

- Certain statements control the flow of execution through the program - if and for are the biggies, but there are others like switch or while, too. They're generally paired with an end statement; anything in between the if or for and the end is under the control of that flow statement. Generally those blocks of code are indented, for readability. They're often nested within each other, too. An if statement specifies some condition to test; if the condition is true, then the block of code before the paired end will be executed. If the condition is false, that code won't run at all. A for statement is used to execute a block of code several times. For statements specify some index variable - often named i or j, but sometimes other things - and specify the range of values it will take on. It'll then loop over the code before the paired end statement and execute it for each value of the index variable. So for i = 1:10 will make a loop that will run 10 times, with i becoming one number bigger each time. This is often used to walk through a whole vector or array of numbers and do something to each element, using the index variable as the index into the array.
- If you get stuck, ask somebody! There are experienced programmers out there who can help.

16.9 Mental Chronometry

Also check out *Jitter* for more info on variable-ISI experiments...

16.9.1 1. What is mental chronometry?

As important (or moreso) than finding out where your activation happened is finding out when it happened. Where did the information flow during the processing of your stimuli? Which structures were active before other structures? Which structures fed output into other structures, and which structures processed end results? Such questions suggest a fairly crude schematic of brain processing, but still a useful one: if you could answer all of those accurately about a given task, tracking the millisecond-by-millisecond flow of information through the brain, you'd have a much fuller picture of the information processing than from a static activation picture. So mental chronometry experiments attempt to attack those questions. First done with reaction time data, using experiments that would add or remove certain stages of a task and find out whether reaction time was sped or stayed the same or what, chronometric experiments have moved into fMRI, with moderate success. Formisano & Goebel (below) review recent developments in fMRI chronometry, with a thorough overview of potential pitfalls. They examine several studies in which flow of information is pinned down with a couple hundred milliseconds - not quite the temporal resolution of EEG, but much better than previously thought could be achieved with fMRI.

16.9.2 2. But how do I get better temporal resolution than my TR?

Simple: don't always sample the sample points of your response. If you always sample the BOLD response 2 seconds and 4 seconds and 6 seconds after your stimuli are presented, for your whole experiment, you'll have a very impoverished picture of the shape of your HRF. But if, for example, you sampled 2 sec. and 4 sec. and 6 sec. post-stimulus for half the experiment, then cut one second between trials and sampled 1 sec. and 3 sec. and 5 sec. for the rest of your experiment - why, then, you'd have a better picture. The cost, of course, is reduced power and expanded confidence intervals at the points you've sampled.

With a good picture of the shape of your HRF, though, you could then compare HRFs from two different regions and see which one had started first, or which one had reached its peak first. If HRF timing is connected in some reliable way to neuronal activation, you then don't need to sample the whole experiment at a super-fast rate - you could infer from only a limited-sample picture of one part of the HRF where neuronal activity had started first and where it had started second, which allows you to rule out certain flows of information.

16.9.3 3. How does variable ISI relate to mental chronometry?

In order to do a mental chronometry experiment, you need to absolutely maximize your statistical efficiency - your ability to pin down the shape of your HRFs. You're going to be comparing HRFs from several regions, and you need to look for the differences between them, which means you can't assume much (if anything) about what shape they're going to be, or else you're going to bias your results. Assuming nothing about your HRF and still getting a good idea of its shape, as Liu describes, means you need an experiment with very high efficiency - and, as Dale demonstrates, those are precisely those designs with variable ISIs. Only by randomizing (or pseudo-randomizing) your ISIs can you pack enough trials into an experiment for sufficient power and still have enough statistical flexibility to get a good look at the shape of your HRFs.

16.9.4 4. What are some pitfalls in mental chronometry with fMRI, then?

The big one is a crucial assumption mentioned above: that HRF timing is connected in some reliable way to neuronal activation. We assume you're not interested so much in the HRF for its own sake, but rather as an indicator of neuronal activity. So let's say you get two HRFs, one from visual cortex and one from motor cortex, and the motor HRF starts half a second later than the visual. Is that because neuronal activity started half a second later in motor cortex? Or is it because the coupling between neuronal activity and BOLD response is just slower in motor cortex in general? Or is it because the coupling in that particular subject just happens to be looser for motor than for visual - and maybe it'll be different for your other subjects! Clearly, no matter how good a look you get at your HRF, questions like these will dog your chronometric experiment unless you're careful about validating your assumptions. Several excellent studies have examined the issue of variability of HRF between regions, subjects, and times, and those studies are crucial to check out before drawing conclusions from this sort of data.

Of course, other more mundane issues may well torpedo chronometric conclusions: if you don't have high enough efficiency in your experiment, you won't be able to distinguish one HRF's shape from another with high accuracy, and you'll have a hard time telling which one started first or second anyways.

16.9.5 5. If I want to do a mental chronometry experiment, how should I design it?

Chronometric experiments depend crucially on determining HRF shape. So start with maximizing that - you need a design that will absolutely maximize your efficiency, no matter what the power cost. M-sequence or permuted block designs are good ways to start (see Liu). It should be obvious that with designs like that, you need experimental tasks that will generate fairly reliable activations; your experiment will suffer in terms of power from its focus on finding HRF shape and not using shape assumptions. Choosing a task with a reasonably long latency is also important - even with the best possible design, fMRI noise is such that resolution below a couple hundred milliseconds is simply not possible for now. So if your task only lasts half a seconds, you may not be able to get much information about the chronometric aspects with fMRI. As well, in an experiment like this, having more samples is always better - so you want to have the shortest possible TR. If you can focus your experiment to a smaller segment of the brain than the whole thing, you can get a good number of slices and still have very fast Trs.

One thing to try and avoid in doing chronometry is to toss it in as a fishing-expedition analysis: if your experiment isn't designed with doing chronometric analysis in mind, you'll almost certainly have trouble finding reliable latency differences in your subjects. Unless you've got an eye on this from the start, it's probably not worth doing. But if you do, you can get some pretty sweet looks at the temporal flow of activation and information around your subjects' heads.

16.10 Normalization

16.10.1 1. What is normalization?

Inconveniently, brains come in all different sizes and shapes. But the standard statistical algorithms all assume that a given voxel - say 10, 10, 10 - samples exactly the same anatomical spot in every subject. So you'd really like to be able to squash every subject's brain into exactly the same shape and exactly the same space in your image. That's normalization - it's the process by which pictures of subjects' brains are squashed, stretched and squeezed to look like they're about the same shape

16.10.2 2. How does normalization work?

A lot like realignment. Normalization algorithms, just like realignment algorithms, search for transformations of the source image that will minimize the difference between it and the target image - transformations that, as much as possible, will make the source image 'look like' the target template. The difference is that realignment algorithms restrict themselves to rigid-body transformations - moving and turning the brain, but not changing its shape. Normalization algorithms allow nonlinear transformations as well - these actually change the shape of the brain, squeezing and stretching certain parts and not other parts, to make the source brain 'fit' the target brain. Different types of nonlinear transformations can be applied - some use sine/cosine basis functions, some use viscous fluid models or meshes - but all normalization can be thought of this way.

An important point about normalization is that any algorithm, if allowed to make changes on a fine enough scale, can precisely transform one brain into another, exactly. Sometimes, though, that's not what you want - if you're interested in looking at differences of gray matter in children vs. in adults, you'd like to normalize the general anatomy, but not at such a fine scale you remove exactly the difference you're looking for! Other times, though, you'd love to match up every point in a subject's brain exactly with the identical point in another subject's brain. Care should still be taken, though - normalization algorithms can align structural anatomy precisely, but can't guarantee the subjects' functional anatomies will align perfectly.

16.10.3 3. Why would I want to normalize? What are the drawbacks and/or advantages?

The advantage is simple: Brains aren't all the same size and shape. The simplest and most widespread methods of statistical analysis of brain data is to look each voxel across all your subjects and compare some measure in that voxel. For that method to be reasonable, equivalent voxels in each subject's images should match up with equivalent locations in each subject's brain. Since brain structures can be quite variable in size and shape, we need some way to 'line up' those structures. If you want to do any kind of voxel-based statistical analysis - not just of activation, but also of anatomy, as in voxel-based morphometry (VBM) - across a group, normalizing can largely remove a huge source of error from your data by removing variance between brain shapes.

The disadvantage is just as simple: Like any preprocessing, normalization isn't perfect. It generally requires interpolation, which introduces small errors into the images, and even with normalization, anatomies may not line up as well as you'd hope. It can also be slow - depending on the methods and programs used, normalizing a run of functional images can take hours or days. Still, to use voxel-based statistics, it's a necessary evil...

16.10.4 4. When is it unhelpful to normalize?

If you're running an analysis that's not voxel-based - say, one that's based on region-of-interest-specific timecourses - then normalization makes a lot less sense. An alternative to voxel-based methods is to compare some measure activation in particular structures hand-drawn (or automatically drawn) on individual subjects' images. Since a method like this gets summary statistics out from each subject individually, without requiring that any statistical images be laid on top of each other, normalization is totally unnecessary. Some researchers choose to preprocess their data on two parallel paths, one with normalization and one without, using the non-normalized data for region-of-interest analysis and the normalized for traditional voxel-based methods.

As well, several factors can make normalization difficult. Lesions, atrophy, different developmental stages, neurological disorders, and other problems can make standard normalization impossible. Some of these problems can be easily addressed (see Brett et. al), and some can't be. Anyone using patient populations with significant neurological differences from their normalization templates should be advised to explore the literature on normalizing patients before proceeding.

16.10.5 5. How important is it to align images to AC-PC before normalizing?

This varies between programs. For AFNI and BrainVoyager, it's pretty important. The nonlinear transformations can account for non-aligned images in theory, but if you start the images off in a non-aligned state, the algorithm is more likely to get caught in a local minimum of the search space, and give you strange normalization parameters. If you aren't realigning before normalizing, it's best to make sure to examine the normalized brains afterwards to make sure that your normalization ran okay. SPM's normalization algorithm has a realignment phase built in that runs automatically before the nonlinear transformations are examined, so doing realignment beforehand isn't necessary. It can't hurt, particularly when realigning runs of functional data, and it's still wise to examine the normalized image afterwards as a sanity check...

16.10.6 6. How important is it to make sure your segmentation is good before normalizing to the gray template?

Very important. The gray template contains, in theory, only gray-matter voxels. Normalization algorithms find their transformations by trying to minimize the voxel-by-voxel intensity differences between images, and white matter, CSF and gray matter all have notably different intensity profiles. So if you have left-over fringes of white matter or CSF or occasional speckles of white matter included by error in your gray-matter image, they'll be treated as error voxels even if they're in the right place. The algorithm may still converge to the best gray-matter solution, but you can greatly increase your chances of getting a good gray-matter normalization by making sure your segmentation is clean and only includes gray-matter voxels.

16.10.7 7. Should you use the inplane anatomy or the high-res anatomy to determine parameters?

There's not a perfect answer, but probably if you have a high-res anatomy, you should use it. In theory, the high-res anatomy should provide you a better match, because it has more detail. However, if you have significant head movement between the high-res scan and the functionals, there will be an additional source of error in the high-res (even after realignment) that may not be there in the inplane if there's less movement between the inplane and the functionals. In general, though, the increased resolution of the high-res will probably provide better precision for your normalization parameters. In practice, the difference will probably be small, but every little bit helps...

16.10.8 8. When in my analysis stream should I normalize?

There are two obvious points when you can normalize - a) in the individual subject analysis, before you estimate your model / do your stats, b) after you've done your stats and calculated contrast images for each subject, but before you do your group analysis. In case a), you'll normalize all your functional images (usually after estimating parameters from an anatomical image); in case b), you'll normalize only your contrast images (always after estimating parametes from an anatomical image). In general, the standard is a), but I'm not sure exactly why. One problem with b) might be that interpolation errors are being introduced directly into your summary statistics, rather than in the functional images they're derived from. To the extent that contrast images are less smooth than functional images, this will tend to disadvantage b). As well, those interpolation errors are then going to be averaged over far fewer observations in b) - when you're combining only one contrast image for each person - than in a) - when you're often combining several hundred functional images for each person. Not sure whether this will make much difference, though... This is a test that should be run.

16.10.9 9. How can you tell how good your normalization is?

There are possible automated ways you can use to determine quantifiably how close your normalization has gotten - see Salmond et. al, for one - but, in general, the easiest way to go is just to compare the template image and your normalized image by looking at them side-by-side (or overlaid). Check to make sure that the gross structures and lobes line up reasonably well, and if you have any particular area of interest - hippocampus, V1, etc. - check those in detail to make sure they line up okay. If they don't, you may want to align your source image differently before normalizing, or try normalizing just your gray matter.

16.10.10 9. What's the difference between linear and nonlinear transformations?

Roughly, linear transformations are those that treat the head as a rigid body, and allow it to be transformed only in ways that don't affect its shape or the shape of anything inside it. Rotations, translations, and scaling all fall in this category. Nonlinear transformations are any transformations that don't respect those constraints; these include any transformations that squeeze parts of the image, stretch parts of the image, or generally distort the shape of the head in any way.

16.10.11 10. How do I normalize children's brains? How about aging brains?

There is a monster literature out there on normalizing various non-standard brains, too large to survey easily; the Wilke et. al is a good start for children and contains some good citations into that literature, and the Brett et. al contains some similarly nice citations for aging brains. Anyone who knows this literature well is invited to contribute links and/or citations...

16.11 P threshold

16.11.1 1. What is the multiple-comparison problem? What is familywise error correction (FWE)?

To start, Nichols and Hayasaka provide an excellent introduction to the issue of FWE in neuroimaging in very readable fashion. You're encouraged to check it out.

Many scientific fields have had to confront the problem of assessing statistical significance in the context of multiple tests. With a single statistical test, the standard conventionally dictates a statistic is significant if it is less than 5% likely to occur by chance - a p-threshold of 0.05. But in fields like DNA microassays or neuroimaging, many thousands of tests are done at once. Each voxel in the brain constitutes a separate test, which usually means tens of thousands of tests for a given subject. If the conventional p-threshold of 0.05 is applied on a voxelwise basis, then, just by chance you're almost guaranteed to have many hundreds of false-positive voxels. In order to avoid any false positives, then, researchers generally correct their p-threshold to account for how many tests they're performing. This type of correction prevents Type I error across the whole family of tests you're doing - a familwise error correction, or FWE correction.

The standard approach to FWE correction has been the Bonferroni correction - simply divide the desired p-threshold by the number of tests, and you'll maintain correct control over the FWE rate. In general, the Bonferroni correction is a pretty conservative correction, and it suffers from a fatal flaw with neuroimaging data. The Bonferroni correction demands that all the tests be independent from each other, and that demand is manifestly not fulfilled in neuroimaging data, where there is a complex, substantial and generally unknown structure of spatial correlations in the data. Essentially, the Bonferroni correction assumes there are more spatial 'degrees of freedom' than there really are; one voxel is not independent from the next, and so one only needs to correct for the 'true' number of independent tests you're doing. This effort, though, is tricky, and so a good deal of theory has been developed on ways around Bonferroni-type corrections that still control the FWE at a reasonable level.

16.11.2 2. What is Gaussian random-field theory and how does it apply to FWE?

Worsley et. al is one of the first papers to link random-field theory with neuroimaging data, and that link has been tremendously productive in the years since. Random-field theory (RFT) corrections attempt to control the FWE rate by assuming that the data follow certain specified patterns of spatial variance - that the distributions of statistics mimic a smoothly varying random field. RFT corrections work by calculating the smoothness of the data in a given statistic image and estimating how unlikely it is that voxels (or clusters or patterns) with particular statistic levels would appear by chance in data of that local smoothness. The big advantages of RFT corrections are that they adapt to the smoothness in the data - with highly correlated data, Bonferroni corrections are far too severe, but RFT corrections are much more liberal. RFT methods are also computationally extremely efficient.

However, RFT corrections make many assumptions about the data which render the methods somewhat less palatable. Chief among these is the assumption that the data must have a minimum level of smoothness in order to fit the theory - at least 2-3 times the voxel size is recommended at minimum, and more is better. For those researchers unwilling to pay the cost in resolution that smoothing imposes, RFT methods are problematic. As well, RFT corrections are only available for statistics whose distributions in a random field have been laboriously calculated and derived - the common statistics fall in this category (F, t, minimum t, etc.), but ad hoc statistics can't be corrected in this manner. Finally, it's become clear (and Nichols and Hayasaka), that even with the assumptions minimally satisfied, RFT corrections tend to be too conservative.

Random-field theory corrections are available by default in SPM; in SPM99 or earlier, choosing a "corrected" p-threshold means using an RFT correction, while in SPM2, choosing the "FWE" correction to your p-threshold uses these methods. I don't believe corrections of this sort are available in AFNI or BrainVoyager.

16.11.3 3. What is false discovery rate (FDR)? How is it different from other types of multiple-comparison correction?

RFT methods may have their flaws, but some researchers have pointed out a different problem with the whole concept of FWE correction. FWE correction in general controls the error rate for the whole family; it guarantees that there's only a 5% chance (for example) of any false positives appearing in the data. This type of correction simply doesn't fit the intuition of many neuroimaging researchers, because it suggests that every voxel activated is a true active voxel, and most researchers correctly assume there's enough noise in every stage of the process to make a few voxels here and there look active just by chance. Indeed, it's rarely of crucial interest in a particular study whether one particular voxel is necessarily truly or falsely positive - most researchers are willing to accept that some of their signal is actually noise - but that level of inference is precisely what FWE corrections attempt to license.

Benjamini & Hochberg, faced with this conundrum, developed a new idea. Rather than controlling the FWE rate, what if you could control the amount of false-positive data you had? They developed a method to control the false discovery rate, or FDR. Genovese et. al recently imported this method specifically into neuroimaging. The idea in controlling the FDR is not to guarantee you have no false positives - it's to guarantee you only have a few. Setting the FDR control level to 0.05 will guarantee that no more than 5% of your active voxels are false positives. You don't know which ones they might be, and you don't even know if fully 5% are false positive. But no more than 5% are falsely active.

The big advantage of FDR is that is adapts to the level of signal present in the data. With small signal, the correction is very liberal. With huge signal, it's relatively more severe. This adaptation renders it more sensitive than an RFT correction if there's any signal present in the data. It allows a much more liberal threshold to be set than RFT, at a cost that most researchers have already mentally paid - a few false positive voxels. It requires almost no computational effort, and doesn't require laborious derivations to be used with new statistics.

FDR is not a perfect cure-all - it does require some assumptions about the level of spatial correlation in the data. At the outer bound, allowing any arbitrary correlation structure, it is only slightly more liberal than the equivalent RFT correction. But with looser assumptions, it's a great deal more liberal. Genovese et. al have argued that fMRI data in many situations fits a very loose set of assumptions, enabling a pretty liberal correction.

The latest edition of every major neuroimaging program provides some methods for FDR control - SPM2 and Brain-Voyager QX have it built-in, and AFNI's 3dFDR program does the same work. Tom Nichols has predicted FDR methods will essentially replace most FWE correction methods within a few years, and they are beginning to be widely used throughout neuroimaging literature.

16.11.4 4. What is permutation testing? How is it different from other types of multiple-comparison correction?

Permutation testing is a form of non-parametric testing, and Nichols and Holmes give an excellent introduction to the field in their paper, a much better treatment than I can give it here. But here's the extreme nutshell version. Permutation tests are a sensitive way of controlling FWE that make almost no assumptions about the data, and are related to the stats/CS concept of 'bootstrapping.'

The idea is this. You hope your experimental manipulation has had some effect on the data, and to the extent that it has, your design matrix is a model that explains the data pretty well, with large beta weights for the conditions of interest. But what if your design matrix had been different? What if you randomly re-labeled your trials, so that a trial that was actually an A trial in the real experiment was re-labeled as a B, and put into the design matrix as a B, and a B trial was re-labeled and modeled as a C trial, and a C as an A, and so forth. If your experiment had a big effect, the new, randomly mixed-up design matrix won't explain it well at all - if you re-ran your model using that matrix, you'd get much smaller beta weights. Of course, on the null hypothesis, there wasn't any effect at all due to your manipulation, which means the random design matrix should explain it just as well.

And now that you've re-labeled your design matrix and re-run your stats, you mix up the design matrix again, differently and do the same thing. And then do it again. And again, until you've run through all the possible permutations of the design matrix (or at least a lot of them). You'll end up with a distribution of beta weights for that condition from possible design matrices. And now you go back and look at the beta weight from your real experiment. If it's at the extreme end of that distribution you've created - congrats! You've got a significant effect for that condition. The idea in permutation testing is you don't make any assumptions about what the statistic distribution could be - you go out and empirically determine it, from your own real data.

But how does that help you with the multiple-comparison problem? One nice thing about permuation testing is that aren't restricted to testing significance for stats with known distributions, like t or F. We can use these on any ad hoc statistic we like. So let's do it across the design matrices, using as our statistic the maximal T: the value of the maximum T-statistic in the whole image for that design matrix. We come up with a distribution, just like before, and we can find the t-statistic that corresponds to the 5% most extreme parts of the maximal T distribution. And now, the clever bit: we go back to our real experiment's statistical map, and threshold it at that 5% level from the maximal T. Hopefully the t-statistics from our real experiment are generally so much higher than those from the random design matrices as to mean a lot of voxels in our real experiment will have t-statistics above that level - and we don't need to correct their significance at all, because anything in that extreme part of the maximal T distribution is guaranteed to be among the most extreme possible t-statistics for any voxel for any design matrix.

Permuation tests have the big advantages of making almost no (but not totally none - see Nichols and Holmes for details) assumptions about your data, which means they work particularly well with low degrees of freedom, where other methods' assumptions about the shape of their statistic's distribution can be violated. They also are extremely flexible - any true or ad hoc statistic can be tested, such as maximal T, or size of structure, or voxel's favorite color - anything. But they have a big disadvantage: computational cost. Running a permutation test involves re-estimating at least 20 models to be able to guarantee a 0.05 significance level, and so in SPM for individual data, that cost can be prohibitive. For other programs, the situation's not as bad, but it can still be pretty difficult to wait. Permuation tests are available at least in SPM99 with the SnPM toolbox, and in AFNI with the 3dMonteCarlo program. Not sure about BrainVoyager.

16.11.5 5. When should I use different types of multiple-comparison correction?

Nichols and Hayasaka's paper does an explicit review of various FWE correction methods (as well as FDR) on simulated and real data of a variety of smoothness levels and degrees of freedom, to judge how conservative or liberal different methods were. Their main findings are:

- Random-field corrections are extremely conservative for all smoothnesses except the highest. This bias becomes stronger as the degrees of freedom go down, such that low-degree-of-freedom, low-smoothness images corrected with RFT methods show the worst underactivation. At the highest smoothness (8-12mm FWHM), they perform reasonably well for all df.
- Permutation methods are almost exact for all degrees of freedom and for all smoothnesses. They become slightly better with data of high smoothness, but basically perform tremendously well under all conditions.
- FDR is not strictly speaking intended to control FWE, but it does an excellent job doing so for low-smoothness data at all degrees of freedom. At high smoothnesses (6mm FWHM and greater), the correction becomes too conservative.

Accordingly, the nutshell recommendations are as follows:

- Random-field methods are good for highly-smoothed data only and are best for single-subject data. For researchers who need a good deal of smoothing to collect significant signal, or who aren't particularly interested in very fine resolution, RFT corrections are quite exact and easily implemented for single subjects. At low degrees of freedom for any smoothness (say, less than 20 df), the RF corrections are generally too conservative for any smoothness.
- For unsmoothed (or low-smoothed), single-subject data, FDR corrections are the best. They have very high sensitivity while still providing good control of false positives, even with low degrees of freedom. Group data tend naturally to be smoother than single-subject data, due to the blurring imposed by anatomical variability, and so may not be ideal for FDR corrections.
- Permutation tests are optimized for group data - they perform perfectly at very low degrees of freedom, where other methods' assumptions are invalidated, and they improve slightly with high-smoothness data, although they still do fine with unsmoothed. In group testing, the permutation is whether each subject's t-statistic signs are true or flipped - presumably, if the mean is zero, flipping the sign of the statistic won't make a difference, but if the mean is nonzero, that flipping will matter. As well, the relative speed of estimating group models in most programs helps counter the increased computational cost of permutation testing in general.

16.11.6 6. What is small-volume correction?

All the FWE correction methods here adapt to the number of tests performed. The fewer tests, the less severe the correction, and in neuroimaging, the number of tests performed corresponds to the number of voxels or the volume corrected. So it's to your advantage when doing FWE correction to minimize the volume you're testing. If you have an a priori hypothesis about where you might see activation, like a particular anatomical structure or a particular area found to be active in another study, you might restrict your correction to only that area and be perfectly valid in only performing FWER correction there. In practice, this is often done when a particular activation is above the uncorrected threshold, but you'd like to report corrected statistics. You might also try it when you're using a corrected threshold to start, but not seeing any activation where you might expect some - you could restrict your correction to a smaller volume than the whole brain and suddenly get activation popping up above the new, small-volume-corrected threshold.

SPM has a shortcut to this sort of volume restriction - the small volume correction (or S.V.C.) button in the results interface. It'll let you re-calculate corrected p-statistics for a specified region only - an ROI mask image, or a sphere around a point, etc. This change won't change the uncorrected p-statistics for any activations, but it will make the corrected p-statistic for any activations in that region significantly better, depending on how big your specified region is.

Note that if you're using an uncorrected threshold to start, using S.V.C. won't show you anything new. This correction only re-jiggers the corrected p-statistic for a given region.

16.11.7 7. What do all the different reported values in my SPM table mean (p-corrected, p-uncorrected, cluster, set, etc.)? How are they calculated?

SPM reports a pair of p-statistics for each voxel, a p-statistic for each cluster, and a p-statistic for each set. At the voxel level, these are relatively self-explanatory. The p-uncorrected statistic is the probability that, by itself, a voxel with that t- (or F-)statistic would occur just by chance. This is the statistic that's used to threshold the brain using the uncorrected threshold, or "None" correction in SPM2. The p-corrected statistic is the probability of that same t-statistic, but corrected for FWE using Gaussian RFT methods. This statistic reflects the volume that's being corrected (and hence changes in small-volume-corrected regions).

The cluster and set values are more obscure and less useful - they're explained in detail in the Friston et. al. Briefly, the cluster-level p-statistic is the probability that a cluster of that size would occur just by chance in data of the given smoothness. The key difference is that the activation of a cluster doesn't imply that any particular voxel in the cluster is active - you can't use that statistic to license inference that any one voxel in the cluster is above some threshold. The set-level p-statistic is similar, at the level of the whole brain; it's the probability that a pattern of activation of that size (number of clusters) would occur in data of the given smoothness. But it doesn't mean that any given cluster is active - it only tells you that there's some particular pattern of activation happening, in a regionally unspecific manner. Because both of these statistics are derived from Gaussian RFT theory, they're both, by definition, corrected p-statistics. But because neither of them license inference to any particular voxel, they're not widely used or cited.

16.11.8 8. What should my p-threshold be for some analysis X?

$p < 0.05$, corrected, remains the gold standard for any neuroimaging analysis. Because RFT corrections are so severe, though (and because other methods aren't widespread enough to challenge them), a de facto standard of $p < 0.001$ seems to be in operation these days a lot of the time. Depending on the type of analysis, you may be able to go even looser - group-level regressions are sometimes seen more loosely, such as $p < 0.005$, although there's not a particularly good reason for this.

Using FDR control instead of FWE correction is relatively new, so by default an FDR of 0.05 seems to be the current standard, but Benjamini & Hochberg, among others, have argued that a more liberal threshold in some situations may be reasonable - as high as 0.1 or even a bit higher.

For any type of non-voxel-based analysis, such as correlations of beta weights, etc., $p < 0.05$ is still the magic number for most reviewers.

16.11.9 9. What should my p-threshold be for conjunction analyses?

A good question. Check out the conjunction papers for more detail, but the basic argument is simple. If a voxel that's active in a conjunction analysis simply has to be active in all of the component analyses, and you're thresholding the conjunction (not any component analyses), then the component analyses should have lower thresholds than the conjunction. Specifically, if you wanted to threshold the conjunction at $p < 0.001$, and you had two components to the conjunction, then you should threshold each of the components at sqrt(0.001). Any voxel active in both of those at that level will be less likely in the conjunction, so you can threshold each component at a very liberal level and be sure the conjunction's threshold will be quite stringent. In short - the conjunction threshold is the product of the component thresholds.

Many researchers, however, disagree with this line of reasoning. First, obviously, this argument depends on all of the components being independent - if they're dependent at all, then the product of the individual thresholds will be more stringent than the true conjunction threshold. Even if they're all independent, though, it's clear that using this line of argument means that any active voxel in the conjunction is a voxel that may well not be active at a "reasonable" threshold in any of the components. This problem is exacerbated with more than two components - with three, say, each component could be thresholded at $p < 0.1$ uncorrected, and the conjunction could have a threshold of $p < 0.001$. This flies in the face of what many people try to argue about their conjunctions, which is that they represent areas that are activated in all of their components. So many researchers use the strategy of simply thresholding their individual components at some liberal but reasonable threshold - $p < 0.001$, or $p < 0.005$ - and then simply assess the intersection of the active areas as the conjunction. This clearly results in extremely significant p-statistics in the conjunction, but it at least gets closer to the idea of "conjunction" that most researchers seem to have.

16.11.10 10. What should my p-threshold be for masked analyses?

If you're masking your one analysis with the results of an another analysis, you're basically doing a conjunction (see above), so you can liberalize your threshold at least a bit. If you're masking your analysis with a region of interest mask, anatomical or otherwise, you might also consider using a small volume correction and using $p < 0.05$ corrected as a threshold. If you're doing some other crazy kind of mask... well, you're kind of in uncharted waters. Start with something reasonable and go from there, and good luck to you.

16.12 Percent Signal Change

Check out *ROI* for more info about region-of-interest analysis in general...

16.12.1 1. What's the point of looking at percent signal change? When is it helpful to do that?

The original statistical analyses of functional MRI data, going way back to '93 or so, were based exclusively on intensity changes. It was clear from the beginning of fMRI studies that raw intensity numbers wouldn't be directly comparable across scanners or subjects or even sessions - average means of each of those things varies widely and arbitrarily. But simply looking at how much the intensity in a given voxel or region jumped in one condition relative to some baseline seemed like a good way to look at how big the effect of the condition was. So early block experiments relied on averaging intensity values for a given voxel in the experimental blocks, doing the same for the baseline block, and comparing the two of 'em. Relatively quickly, fancier forms of analysis became available, and it seemed obvious that correcting that effect size by its variance was a more sensitive analysis than looking at it raw - and so t-statistics came into use, and the general linear model, and so forth.

So why go back to percent signal change? For block experiments, there are a couple reasons, but basically percent signal change serves the same function as beta weights might (see *ROI* for more on them): a numerical measure of the effect size. Percent signal change is a lot more intuitive a concept than parameter weights are, which is nice, and many people feel that looking at a raw percent signal change can get you closer to the data than looking at some statistical measure filtered through many layers of temporal preprocessing and statistical evaluation.

For event-related experiments, though, there's a more obvious advantage: time-locked averaging. Analyzing data in terms of single events allows you to create the timecourse of the average response to a single event in a given voxel over the whole experiment - and timecourses can potentially tell you something completely different than beta weights or contrasts can. The standard general linear model approach to activation assumes a shape for the hemodynamic response, and tests to see how well the data fit that model, but using percent signal change as a measure lets you actually go and see the shape of the HRF for given conditions. This can potentially give you all kinds of new information. Two voxels might both be identified as "active" by the GLM analysis, but one might have an onset two seconds before the next. Or one might have a tall, skinny HRF and one might have a short but wide HRF. That sort of information may shed new light on what sort of processing different areas are engaging in. Percent signal change timecourses in general also allow you to validate your assumptions about the HRF, correlate timecourses from one region with those from another, etc. And, of course, the same argument about percent signal change being somehow "closer" to the data still applies.

Timecourses are rarely calculated for block-related experiments, as it's not always clear what you'd expect to see, but for event-related experiments, they're fast becoming an essential element of a study.

16.12.2 2. How do I find it?

Good question, and very platform dependent. In AFNI and BrainVoyager, whole-experiment timecourses are easily found by clicking around, and in the Gablab the same is available for SPM with the Timeseries Explorer. Peristimulus timecourses, though, ususally require some calculation. In SPM, you can get fitted responses through the usual results panel, using the plot command, but those are in arbitrary units and often heavily smoothed relative to the real data. The simplest way these days for SPM99 is to use the Gablab Toolbox's roi_percent code. Check out RoiPercent for info about that function. That creates timecourses averaged over an ROI for every condition in your experiment, with a variety of temporal preprocessing and baseline options. In SPM2, the new Gablab roi_deconvolve is sort of working, although it's going to be heavily updated in coming months. It's based off AFNI's 3dDeconvolve function, which is the newest way to get peristimulus timecourses in AFNI. That's based on a finite impulse response (FIR) model (more on those below). BrainVoyager's ROI calculations will also automatically run an FIR model across the ROI for you.

16.12.3 3. How do those timecourse programs work?

The simplest way to find percent signal change is perfectly good for some types of experiments. The basic steps are as follows:

- Extract a timecourse for the whole experiment for your given voxel (or extract the average timecourse for a region).
- Choose a baseline (more on that below) that you'll be measuring percent signal change from. Popular choices are "the mean of the whole timecourse" or "the mean of the baseline condition."
- Divide every timepoint's intensity value by the baseline, multiply by 100, and subtract 100, to give you a whole-experiment timecourse in percent signal change.
- For each condition C, start at the onset of each C trial. Average the percent signal change values for all the onsets of C trials together.
- Do the same thing for the timepoint after the onset of each C trial, e.g., average together the onset + 1 timepoint for all C trials.
- Repeat for each timepoint out from the onset of the trial, out to around 30 seconds or however long an HRF you want to look at.

You'll end up with an average peristimulus timecourse for each condition, and even a timecourse of standard deviations/confidence intervals if you like - enough to put confidence bars on your average timecourse estimate. This is the basic method, and it's perfect for long event-related experiments - where the inter-trial interval is at least as long as the HRF you want to estimate, so every experimental timepoint is included in one and only one average timecourse.

This method breaks down, though, with short ISIs - and those are most experiments these days, since rapid event-related designs are hugely more efficient than long event-related designs. If one trial onsets before the response of the last one has faded away, then how do you know how much of the timepoint's intensity is due to the previous trial and how much due to the current trial? The simple method will result in timecourses that have the contributions of several trials (probably of different trial types) averaged in, and that's not what you want. Ideally, you'd like to be able to run trials with very short ISIs, but come up with peristimulus timecourses showing what a particular trial's response would have been had it happened in isolation. You need to be able to deconvolve the various contributions of the different trial types and separate them into their component pieces.

Fortunately, that's just what AFNI's 3dDeconvolve, BrainVoyager QX, and the Gablab's roi_deconvolve all do. SPM2 also allows it directly in model estimation, and Russ Poldrack's toolbox allows it to some degree, I believe. They all use basically the same tool - the finite impulse response model.

16.12.4 4. What's a finite impulse response model?

Funny you should ask. The FIR model is a modification of the standard GLM which is designed precisely to deconvolve different conditions' peristimulus timecourses from each other. The main modification from the standard GLM is that instead of having one column for each effect, you have as many columns as you want timepoints in your peristimulus timecourse. If you want a 30-second timecourse and have a 3-second TR, you'd have 10 columns for each condition. Instead of having a single model of activity over time in one column, such as a boxcar convolved with a canonical HRF, or a canonical HRF by itself, each column represents one timepoint in the peristimulus timecourse. So the first column for each condition codes for the onset of each trial; it has a single 1 at each TR that condition has a trial onset, and zeros elsewhere. The second column for each condition codes for the onset + 1 point for each trial; it has a single 1 at each TR that's right after a trial onset, and zeros elsewhere. The third column codes in the same way for the onset + 2 timepoint for each trial; it has a single 1 at each TR that's two after a trial onset, and zeros elsewhere. Each column is filled out appropriately in the same fashion.

With this very wide design matrix, one then runs a standard GLM in the multiple regression style. Given enough timepoints and a properly randomized design, the design matrix then assigns beta weights to each column in the standard way - but these beta weights each represent activity at a certain temporal point following a trial onset. So for each condition, the first column tells you the effect size at the onset of a trial, the second column tells you the effect size one TR after the onset, the third columns tells you the effect size two TRs after the onset, and so on. This clearly translates directly into a peristimulus timecourse - simply plot each column's beta weight against time for a given condition, and voila! A nice-looking timecourse.

FIR models rely crucially on the assumption that overlapping HRFs add up in linear fashion, an assumption which seems valid for most tested areas and for most inter-trial intervals down to about 1 sec or so. These timecourses can have arbitrary units if they're used to regress on regular intensity data, but if you convert your voxel timecourses into percent signal change before they're input to the FIR model, then the peristimulus timecourses you get out will be in percent signal change units. That's the tack taken by the Gablab new roi_percent. Some researchers have chosen to ignore the issue and simply report the arbitrary intensity units for their timecourses.

By default, FIR models include some kind of baseline model - usually just a constant for a given session and a linear trend. That corresponds to choosing a baseline for the percent signal change of simply the session mean (and removing any linear trend). Most deconvolution programs include the option, though, to add other columns to the baseline model, so you could choose the mean of a given condition as your baseline.

There are a lot of other issues in FIR model creation - check out the AFNI 3dDeconvolve model for the basics and more.

16.12.5 5. What are temporal basis function models? How do they fit in?

Basis function models are a sort of transition step, representing the continuum between the standard, canonical-HRF, GLM analysis, and the unconstrained FIR model analysis. The standard analysis assumes an exact form for the HRF you're looking for; the FIR places no constraints at all on the HRF you get. But sometimes it's nice to have some kinds of constraints, because it's possible (and often happens) that the unconstrained FIR will converge on a solution that doesn't "look" anything like an HRF. So maybe you'd like to introduce certain constraints on the type of HRFs you'll accept. You can do that by collapsing the design matrix from the FIR a little bit, so each column models a certain constrained fragment of the HRF you'd like to look for - say, a particular upslope, or a particular frequency signature. Then the beta weight from the basis function model represents the effect size of that part of the HRF, and you can multiply the fragment by the beta weight and sum all the fragments from one condition to make a nice smooth-looking (hopefully) HRF.

Basis function models are pretty endlessly complicated, and the interested reader is referred to the papers by Friston, Poline, etc. on the topic - check out the Friston et. al, "Event-related fMRI" paper.

16.12.6 6. How do you select a baseline for your timecourse? What are pros and cons of possible options? Do some choices make particular comparisons easier or harder?

Good question. Choosing a particular baseline places a variety of constraints on the shape of possible HRFs you'll see. The most popular option is usually to simply take the mean intensity of the whole timecourse - the session mean. The problem with that as a baseline is that you're necessitating that there'll be as much percent signal change under the baseline as over it. If activity is at its lowest point during the inter-trial interval or just before trial onset, then, that may lead to some funny effects, like the onset of a trial starting below baseline, and dramatic undershoots. As well, if you've insufficiently accounted for drifts or slow noise across your timecourse, you may overweight some parts of the session at the expense of others, depending on what shape the drift has. Alternatively, you could choose to have the mean intensity during a certain condition be the baseline. This is great if you're quite confident there's not much response happening during that condition, but if you're not, be careful. Choosing another condition as the baseline essentially calculates what the peristimulus timecourse of change is between the two conditions, and if there's more response at some voxels than you thought in the baseline condition, you may seriously underestimate real activations. Even if you pick up a real difference between them, the difference may not look anything like an HRF - it may be constant, or gradually increase over the whole 30 seconds of timecourse. If you're interested in a particular difference between two conditions, this is a great option; if you're interested in seeing the shape of one condition's HRF in isolation, it's iffier.

With long event-related experiments, one natural choice is the mean intensity in the few seconds before a trial onset - to evaluate each trial against its own local baseline. With short ISIs, though, the response from the previous trial may not have decayed enough to show a good clean HRF.

16.12.7 7. What kind of filtering should I do on my timecourses?

Generally, percent signal analysis is subject to the same constraints in fMRI noise as the standard GLM, and so it makes sense to apply much of the same temporal filtering to percent signal analysis. At the very least, for multi-session experiments, scaling each session to the same mean is a must, to allow different sessions to be averaged together. Linear detrending (or the inclusion of a first-order polynomial in the baseline model, for the AFNI users) is also uncontroversial and highly recommended. Above that, high-pass filtering can help remove the low-frequency noise endemic to fMRI and is highly-recommended - this would correspond to higher-order polynomials in the baseline model for AFNI, although studies have shown anything above a quadratic isn't super useful (Skudlarski et. al). Low-pass filtering can smooth out your peristimulus timecourses, but can also severely flatten out their peaks, and has fallen out of favor in standard GLM modeling; it's not recommended. Depending on your timecourse, outlier removal may make sense - trimming the extreme outliers in your timecourse that might be due to movement artifacts.

16.12.8 8. How can you compare time courses across ROIs? Across conditions? Across subjects? (peak amplitude? time to peak? time to baseline? area under curve?) How do I tell whether two timecourses are significantly different? How can you combine several subjects' ROI timecourses into an average? What's the best way?

All of these are great questions, and unfortunately, they're generally open in the literature. FIR models generally allow contrasts to be built just as in standard GLM analysis, so you can easily do t- or F-tests between particular aspects of an HRF or combinations thereof. But what aspects make sense to test? The peak value? The width? The area under the curve? Most of these questions aren't super clear, although Miezin et. al and others have offered interesting commentary on which parameters might be the most appropriate to test. Peak amplitude is the de facto standard, but faced with questions like whether the tall/skinny HRF is "more" active than the short/fat HRF, we'll need a more sophisticated understanding to make sense of the tests.

As for group analysis of timecourses, that's another area where the literature hasn't pushed very far. A simple average of all subjects' condition A, for example, vs. all subjects' condition B may well miss a subject-by-subject effect because of differing peaks and shapes of HRFs. That simple average is certainly the most widely used method, however, and so fancier methods may need some justification. One fairly uncontroversial method might be simply analogous to the standard group analysis for regular design matrices - simply testing the distribution across subjects of the beta weight of a given peristimulus timepoint, for example, or testing a given contrast of beta weights across subjects.

16.13 Physiology and fMRI

This section is intended to address design-related questions that focus primarily on how physiological factors can affect your scanning - things like heart rate, breathing, the types of artifacts generated by those movements, etc. Obviously, physiological factors are mixed in heavily with your experimental design, so be sure to check out some other design-related pages:

- *Design*
- *Scanning*
- *Jitter*

16.13.1 1. Why would I want to collect physiological data?

No, the real question should be: why wouldn't you want to collect physiological data? And the only answer is: because you hate freedom. Haha! Phew.

Actually, the main reason is because physiological effects can be a significant source of noise in your data. The pulsing of blood vessels with the cardiac cycle can move brain tissue, jostle ventricles and alter BOLD signal in specific regions; respiration can induce magnetic inhomogeneities and create head movements. If you measure physiology, you can, at least in part, account for those sources of noise and remove their confounding effects from your data, boosting your signal to noise ratio.

Secondarily (well, primarily, for some), for many studies, physiological measures like heart rate or respiration rate can be an important source of data themselves, providing an important test of autonomic arousal that may supplement self-report data or other measures. All those uses are beyond the scope of this discussion, but it's something to keep in mind.

16.13.2 2. How do I do it?

Thankfully, scanner designers have generally had the foresight to think someone might want to collect this info, and so most scanners have instruments built in to record at least two main physiological measures - heartrate/cardiac cycle (usually with a photoplethysmograph - a small clip that goes on the finger) and respiration (usually with a pneumatic belt - a thin belt that wraps around the chest). For scanners without these instruments built in, several companies manufacture MR-compatible versions of these instruments.

There may be other measures that you may want to collect - galvanic skin response, for example - and here at Stanford, those sensors are relatively easily available. Other measures have less of an effect on creating noise in the signal, however, and so we'll primarily address those two measures.

16.13.3 3. How might the cardiac cycle influence my signal?

Several ways. The pulsation of vessels creates a variety of other movements, as CSF pulses along with it to make room for incoming blood, tissue moves aside slightly as vessels swell and shrink, and waves of blood (and the accompanying BOLD signal) move through the head. This sounds like the effects can be relatively small, but large structures of the brain can move significant amounts in the neighborhood of large vessels, and the resulting motion can significantly change your signal. Generally, the term "pulsatility" is used to describe the process that generates artifacts through cardiac movement. As well, because TRs for many experiments are slower than the cardiac cycle, these effects can occur image-to-image in an unpredictable way, as various points in the cycle are sampled in an irregular fashion.

These effects are more significant in some regions of the brain than in others; Dagli et. al address the question of where pulsatility artifact is worst. Perhaps unsurprisingly, areas near large vessels tend to rank among those regions, but other areas are also affected - regions near the borders of are also significantly affected.

16.13.4 4. How might respiration influence my signal?

Also a couple ways. Respiration, by its nature, can cause the head and particular parts of it (sinus cavities, etc.) to move slightly, which can induce motion-related changes in signal. Perhaps more significantly, the inflating and deflating of the lungs changes the magnetic signature of the human body, and that signature change can induce inhomogeneities in the baseline magnetic field (B0) that you've carefully tuned with your shim. Those inhomogeneities can be unpredictable and can affect your signal in unpredictable ways. Breathing rate can also be significantly less predictable than cardiac cycle - many subjects take spontaneous deeper breaths at irregular intervals, for example. Van de Moortele et. al address the sources of respiration artifact in some detail.

16.13.5 5. What can I do to account for these changes?

Thought you'd never ask. There are several ways. Pfeuffer et. al present a navigator-based method in k-space that adjusts, in large part, for global effects, more due to respiration changes. Perhaps the most prevalent ways, though, are retrospective, and rest on the fact that generally, cardiac cycle and respiration cycle take place at a time scale far faster than stimulus presentation for most experiments. Isolating signal changes that take place at the appropriate frequency, then, can help isolate those sources of noise.

This isn't as easy as it sounds, due to the potential aliasing of this noise because of the difference between TR and cardiac cycle time. But it is possible, with some work. Glover et. al present one of the industry-standard ways of doing this correction - by sorting images according to their point in the cardiac or respiration cycle, the appropriate amount of signal change due to those sources can be identified and removed from the image. This correction happens at the point of reconstruction of images from raw data, and is available automatically at Stanford in the makevols program.

Care should be taken with this option, though. As with any type of algorithm that removes "confounding" signals (realignment, say), this correction can't account for the extent to which physiological noise is correlated with the task. If there is significant correlation between your task and your physiological measures, removing noise due to the physiological sources will also remove task-related signal. This could happen with rapid event-related tasks if subjects breath in time with the tasks, or in any sort of experiment that might induce arousal - emotionally arousing stimuli may increase heart rate and respiration rate and thus change those noise profiles in a task-correlated way. In this case, Glover et. al suggest using only resting-state images to calculate this correction; this is always a significant consideration in physiological artifacts.

16.13.6 6. Are there other sources of physiological noise I might want to worry about?

Possibly. Peeters and Van der Linden address the question of longer-term physiological changes, such as those induced by pharmacological manipulations or sudden environmental changes. As a drug begins to be absorbed, for example, there can be a gradual change in vasoconstriction or blood oxygenation in general that can look like a global signal drift but is, in fact, a changing of the sources of the signal. Researchers interested in looking at these sorts of changes should look with care at these sorts of corrections.

16.14 ROI

16.14.1 1. What's the point of region-of-interest (ROI) analysis? Why not just use the basic voxelwise stats? Are you too good for them or something? Huh, Mr. Fancy Pants?

Whoa, now, no need to get touchy. A lot of people think ROI analysis is a really good idea, possibly where the real future of fMRI lies. Nieto-Castanon et. al make the argument kind of like this: brain imaging is concerned, among other things, with analyzing how mental functions are connected to brain anatomy. Note that the word "voxel" didn't enter into that statement. Voxels aren't, on their own, a particularly useful concept for us, but the standard statistical model does all of its preprocessing and analysis on that level. That analysis path tends to completely blur anatomical boundaries, often by a great deal, smearing our resolution to hell and preventing us from making good clean associations between structure and function. So ROI analysis offers us a way to get around individual anatomical variability and sharpen our inferences.

As well, even if we don't start our statistical analysis at the ROI level, ROIs offer us a reasoned way to extract measures that differ from the standard voxelwise t-statistic. Measures like percent signal change timecourses or fit coefficients are additional information that you can extract from your data only by looking within a particular region. These measures can shed light on otherwise obscured aspects of your study - temporal characteristics, particular sizes and directions of effects, or correlations with behavior. Seen from this perspective, ROI analysis is a valuable parallel tool to the standard voxelwise GLM analysis and can often provide new and interesting pieces of the puzzle of your data.

16.14.2 2. How should I generate ROIs? What are the pros and cons of each way?

Funny you should ask; I just happened to have this little grid lying around, which has been helpfully converted into a sort of tree for the Digi-Web. The methods of ROI definition can be split along two axes - the type of brain ROIs are defined on (individual, group, atlas) and the features used to define it (microanatomy/cytoarchitecture, macroanatomy, function). The breakdown goes like this:

Microanatomy / cytoarchitecture ROIs

- Individually-defined: This would mean drawing (by hand or automated method) ROIs on the cytoarchitectonic map for an individual. The pros of this method: you're quite close to the neuronal level of organization, you've got very high resolution on your ROI, and the mapping between cytoarchitectonic regions has generally proven to be close (see Brett et. al). Cons: Not all functional info is represented (columnar structure, for example, is sometimes represented in cytoarchitecture and sometimes not), and, of course, getting the cytoarchitectonic map for a living human non-invasively is practically impossible at this point. That may change in the future, though - some groups are attempting to mine this data from fMRI maps.
- Group-defined: This is almost never done - it would mean drawing cytoarchitectonic maps on a template or something.
- Atlas-defined: This means getting coordinates for particular cytoarchitectonic regions from an atlas, and generally this means using the Talairach Daemon or some similar tool to get Brodmann Areas for particular atlas coordinates. Pros of this: the Tal Daemon's easy to get, easy to use, fast, popular, and can be easily used for MNI-space coordinates with a simple transform. Cons: Lots of sources of error in labeling. The original Talairach BA labeling is very crude (see Brett et. al) and based on eyeball. The Talairach - MNI transform isn't perfect. And perhaps biggest of all, there is enormouse individual variability of the shape and size of cytoarchitectonic regions - what's BA32 on the atlas may well be deep into BA10 on your subject. Some of

those problems are solved by getting a better cyto. atlas - Zilles et. al are working on one - but the problem of individual variability remains.

Macroanatomy / anatomically-defined ROIs

- Individually-defined: This means drawing ROIs on each individual subject's anatomy, based on anatomical markers like sulci, gyri, and other features viewable by the naked eye. This drawing can be by hand or by an automated or semi-automated program . (See *Segmentation* for more info on those - they're getting pretty good.) Pros: Gets around problem of individual anatomical variability - by extracting each measure from an individually defined structure, you eliminate the issue of whether everyone's structures line up. Allows you to keep your data unsmoothed (smoothing is often done to avoid problems of anatomical variability), and so gives you higher resolution without sacrificing power and even while gaining some. Better mapping of function to structure. Cons: Can be hard to get - requires either good use of an automated algorithm or a lot of labor and experience in hand-drawing ROIs. Makes it more difficult to report activation coordinates (but note Swallow et. al suggest you can normalize your data before defining and be okay). Most importantly, the relationship between function and macroanatomy is very unclear in many regions of the brain, particularly associational cortex and prefrontal areas. Just because you're extracting from everybody's superior frontal gyrus doesn't mean you're going to get at all the same function. So this path may not gain you any power at all and might lose you some.
- Group-defined: Again, this is rarely done - it would mean drawing your ROI on some group average. Probably all of the cons of above, with fewer pros.
- Atlas-defined: This means taking some atlas system's description of anatomical features, like using the Talairach-defined amygdala or Talairach superior frontal gyrus, or possibly drawing an ROI on the MNI template brain. Often the ROIs are then reverse-normalized to fit the subject's non-normalized functional data. Pros: Very easy to get - probably the most-used method for defining anatomical ROIs. Very standardized, and so easily comparable across studies. There's not as much an issue with labeling as above - the Talairach and Tournoux atlas is very accurate at picking out coordinates for macroanatomical features, so you get to leverage their skill at drawing ROIs for your own study. Cons: The overlap of the atlas-defined region with your subjects' is only as good as your normalization - and Nieto-Castanon et. al offer a scathing commentary on how good standard normalization algorithms are, showing that even post-normalization, they had huge variations in the shape of particular gyri. Because of that, atlas-defined regions will always tend to obscure differences in anatomical variability, even with reverse normalization. As well, this method is still subject to the problem of unclear structure-function relationships in many areas of the brain.

Functional ROIs

- Individually-defined: This means choosing functionally-activated voxels from each individual's results - either from a localizer task or from the actual study task. This is so popular these days we've actually broken it out to its own page, FunctionalLocalization.
- Group-defined: This means choosing functionally-activated voxels from your group results and using that voxel set as an ROI to extract from individuals. Pros: Gets to function, as above. Can be a better guarantee that you've got some effect in a given region, since presumably voxels activated in the group had some decent activations in most or all subjects. A good chunk less labor than individually-defining ROIs. Cons: Swallow et. al demonstrate these ROIs are not as reliable (i.e., are more variable) than the individually-created ones. This method doesn't account at all for individual functional variability, which can be considerable (i.e., different voxels can be included or exluded wrongly in every subject).
- Atlas-defined: Probably the closest thing to this is using some other study's reported sites of functional activation - choosing ROIs that correspond to the Tal or MNI coords of other studies' activations. Pros: Avoids the problem of defining localizers, allows direct comparison between studies. Cons: Ignores differences in subject pool and anatomical and functional variability in your own subjects. Also introduces whatever error the other study might have had in localization into your study - if they got their spot wrong, you might be even wrong-er...

Lotta options. All of 'em have their pros and cons - which one you choose will depend largely on the type of questions you want to ask.

16.14.3 3. When can I just look at peak voxels vs. whole regions?

This is still an open question in the literature. The argument for averaging across an ROI is that it should enhance signal to noise; the timecourse from a single voxel can be quite noisy and could, indeed, be some kind of outlier in the ROI. Averaging might give you a better picture of what's happening over the whole ROI. The argument for using a peak voxel, though, is that we know the peak voxel - the voxel that shows the most correlation to the task relative to its variance - is guaranteed to show the best effect of any voxel in the ROI. Additionally, since we know our resolution is blurred by the vascular structure in the region, any spatial smoothing we may have done, and registration and normalization errors, it's entirely possible that some of our ROI's activation isn't reflecting "true" neuronal activity but simply an echo or blurring of activity elsewhere in the ROI. So to average those timecourse together may well wash out our effect, which is after all calculated in voxelwise fashion.

Nieto-Castanon et. al choose to look at whole ROIs, and that's arguably the prevailing sentiment in the literature. Particularly with false discovery rate p-threshold correction rising in prominence (see *P threshold*), the risk of any given voxel being a false positive might seem too high.

On the other hand, at least one (and possibly more than one) empirical study - Arthurs & Boniface - has found that peak-voxel activity correlates better with evoked scalp electrical potentials than does activity averaged across an ROI. They cite a couple other studies that have examined similar issues in animal models, and suggest that in mammal cortex in general, the brain may "water the garden for the sake of one thirsty flower," i.e., ROI activity may only reflect true neuronal changes in a few voxels of the ROI. So the question remains open...

16.14.4 4. What sort of measures can I get out of ROIs?

The two big ones that are usually looked at are:

- beta weights (also called parameter weights or fit coefficients), which are voxel-by-voxel slope values from the multiple regression of your statistical model and correspond with effect sizes of particular conditions.
- percent signal change or timecourse information - literally looking at the TR-by-TR image intensity at a particular voxel or ROI to get a timecourse of intensities in a particular area. That timecourse is often trial-averaged to come up with time-locked average timecourse, which correspond to the average intensity change following the onset of a particular trial type - essentially an empirical look at the shape of the hemodynamic response to different trial types in a given region.

Other measures are occasionally taken - for voxel-based morphometry, for example, where you might look at percentage of gray matter in a given voxel - but those two are the biggies. Beta weights are used in block-related and event-related experiments; percent signal change is usually more important for event-related experiments, although it's occasionally used for blocks as well.

16.14.5 5. What's the point of looking at percent signal? When is it helpful to do that? How do I find it?

For everything you could want to know about percent signal change, check out *Percent Signal Change*.

16.14.6 6. What are beta weights / parameter weights / fit coefficients? When is it helpful to look at them? What types of analyses can I do with them?

When you run a general linear model to estimate your effect sizes (see *Basic Statistical Modeling* for info on this), you're essentially running a giant multiple regression on your data, with the columns of your design matrix as the regressors. Each of those columns corresponds to a particular effect, and each of them is assigned by the GLM a particular parameter value: the B in the equation Y = XB + E, where Y is the signal, X is the design matrix, and E is error. That parameter value corresponds to how large an effect the particular condition had in influencing brain activity.

Importantly, beta weights are not an index of how well your condition's design matrix fit the brain activity - it is not the r or r-squared value for the regression. It's the slope of the regression. This means you could conceivably have a very small effect that fit the model incredibly well, or a very large effect with a great deal of noise in the response. This slope corresponds better to the idea of 'level of activation for a particular condition' that we want to find. As an example, a design matrix column that was all zeros might predict the brain activity perfectly - there might be essentially no change in a given voxel down the whole timecourse. In that case, our r-squared would be very high for that column of the regression - but we wouldn't want to say that voxel was active, because it was totally insensitive to any experimental manipulation. It makes more sense to look at how big the effect size was - whether a given voxel seemed to respond very highly to a given trial type - and then, if we're concerned about noise, we can normalize the effect size by some measure of the effect variance, to get a t-statistic. That's generally what's done in most neuroimaging programs these days.

The big reason to extract beta weights at all is that they give you a numerical estimate of the effect size of a particular condition. If the beta for A at a given point for one subject is three times that at the same point for another subject, you know that A had three times a bigger effect in the first subject. This can be an ideal measure to use in regressions against some behavioral measure. For example, you might want to know if a given's subject's self-reported difficulty with a task correlated with the size of the effect of that condition in a particular region. You can extract the beta weights from that regions for each subject, run a simple regression, and find how significant it comes out.

You can also use beta weights to correlate with each other as a crude way of indexing connections between regions. If subjects with anterior cingulates that responded more to condition A also had cerebellums that responded less to condition A, and that correlation is significant across your subject pool, that may tell you something about how the cingulate and cerebellum relate in your task. Check out *Connectivity* for more info that direction.

Essentially, beta weights can be used in a myriad of ways - any time you'd like to have some numerical estimate of a given effect or contrast size, rather than simply a statistical measure of the activation.

16.14.7 7. How do I find beta weights / etc.?

Depends on the program, but every major neuroimaging program can create, in the process of running a GLM, an image of the voxel-by-voxel beta weights for each condition. In SPM, these are the beta_00*.img files produced by estimating a model; in AFNI, they're the fit-coefficient or parameter images that can be produced as part of the bucket dataset output. Other programs generally have similar names for these images. Getting the beta weights is simply a matter of using some image extraction utility - something like roi_extract for SPM - to get the voxel-by-voxel intensity values in your desired ROI. These can then be averaged across the ROI, or you can look only at the peak beta value, etc. Check out *ROI* for more.

16.14.8 8. How do I combine information from an ROI across the whole thing?

The most common strategy in dealing with multiple voxels in an ROI is simply to average your measure across all voxels in the ROI. This has the advantage of being simple to do and simple to explain in a paper. Friston et al. (2006) point out this method may be too conservative; if the ROI has any heterogeneity (say, half of it activates and half of it de-activates), you'll tend to miss things. More complicated methods can be used to identify different subsets of voxels within the ROI with separable responses. Taking the first eigenvariate of the response across voxels from a principal components analysis of the ROI is a simple version of this (and supported in SPM).

16.15 Random and Fixed Effects

16.15.1 1. What is a random-effects analysis? What's a fixed-effects analysis? What's the difference?

Random-effects and fixed-effects analyses are common concepts in social science statistics, so there are a lot of good intros to them out on the web. But in a very small nutshell: A fixed-effects analysis assumes that the subjects you're drawing measurements from are fixed, and that the differences between them are therefore not of interest. So you can look at the variance within each subject all lumped in together - essentially assuming that your subjects (and their variances) are identical. By contrast, a random-effects analysis assumes that your measurements are some kind of random sample drawn from a larger population, and that therefore the variance between them is interesting and can tell you something about the larger population.

Perhaps the most fundamental difference between them is of inference. A fixed-effects analysis can only support inference about the group of measurements (subjects, etc.) you actually have - the actual subject pool you looked at. A random-effects analysis, by contrast, allows you to infer something about the population from which you drew the sample. If the effect size in each subject relative to the variance between your subjects is large enough, you can guess (given a large enough sample size) that your population exhibits that effect - which is crucial for many group neuroimaging studies.

16.15.2 2. So what does the difference between them mean for neuroimaging data?

If you're interested in making any inferences about the population at large, you essentially are required to do some kind of random-effects analysis at some point in your stream. Not all studies demand this - some types of patient studies, for example - but in general, a random-effects analysis will take place at some point. However, random-effects analyses tend to be less powerful for neuroimaging studies, because they only have as many degrees of freedom as number of subjects. In most neuroimaging studies, you have vastly more functional images per subject than you do subjects, and so you have vastly more degrees of freedom in a fixed-effects analysis.

16.15.3 3. In what situations are each appropriate for neuroimaging analysis?

Generally, a neuroimaging study with more than one or two subjects will have a place for both types of analysis. The typical study proceeds with a type of model called the hierarchical model, in which both fixed and random effects are considered, but the two types of factors are limited and entirely separable. Single-subject analyses are generallly carried out with a fixed-effects model, where only the scan-to-scan variance is considered. Those analyses generally yield some type of summary measure of activation, be it a T-statistic or beta weight or other statistic. Once those summary measures are collected for each subject, then, a random-effects analysis can be performed on the summaries, looking at the variance between effect sizes as a random effect. Again, only a single source of variance is considered at a single time.

For the most part, the rule of thumb is: single-subject analyses should be fixed-effects (to leverage the greater power of a fixed-effects model) and any analysis involving a group of subjects that you'd like to express something about the population should be random-effects.

16.15.4 4. How do I carry out a fixed-effects analysis in AFNI/SPM/BrainVoyager?

Generally, the standard single-subject model in all neuroimaging software is a fixed-effects model. Only a single source of variance is considered - the variance between scans (or points in time). If you include several subjects' functional images in a single fMRI model (as opposed to basic model) in SPM, for example, the program will run fine - you'll just get a fixed-effects model over several subjects at once. Any program that produces summary statistic images from single subjects will generally be a fixed-effects model: the standard GLM analysis in SPM and BrainVoyager, for example, or 3dFIM+ or 3dDeconvolve in AFNI. All of these apply a fixed-effects model of your experiment to look at scan-to-scan variance for a single subject. Other subjects could be included, as mentioned, but the variance between subjects will not generally be considered.

16.15.5 5. How do I carry out a random-effects analysis in AFNI/SPM/BrainVoyager?

Until a few years ago, this was a trickier question, but the Holmes & Friston paper highlighted the need for random-effects models in group neuroimaging studies, and since then (and before, in some cases), every major neuroimaging program has made the hierarchical model the default for group analysis. The idea is built into every program and quite simple: once you've got summary images of the effect sizes from each of your subjects (from single voxels or ROIs or whatever), you then simply throw those effect size summaries into a 'basic' statistical test to look for effect size across the effect sizes. The simplest is a one-sample t-test, but more complicated models can also be used: regressions, ANOVAs, etc. In SPM and BrainVoyager, the 'basic models' button or menu will take you to these sorts of group tests; in AFNI, 3dttest is a simple group t-test program, or 3dregana will do group regressions.

16.15.6 6. Which files should I include in my random-effects analysis? Contrast images? T-statistic images? F-statistics? Why one and not the others?

This is an important point, and explained better by Holmes' random-effects model, which should be required reading for anyone doing a random-effects test. In general, you want to include whatever image is a summary of your effect size, and not a measure of the significance of your effect size. Evaluating the significance of a group of significances is a layer beyond the statistics you're interested in - you want your measurements to really reflect how big the effect was at the ROI or voxel, not anything about the rest of the variance across the brain. So in general, for a t-test contrast, the image you want to include is the contrast image - the weighted sum of your beta weights - or a raw beta image. In SPM, that's the con_00*.img files, or the beta files themselves.

As a side note, for tests of more than one constraint at once - such as F-tests - the proper summary image is actually kind of tricky - simply including the ESS (Extra Sum of Squares) image into a standard random-effects test is not the way to go. SPM has a multivariate toolboox that may be of help in handling group F-tests directly, but more usually, the approach is to figure out what constraint in the F-test is driving the effect and use that constraint's contrast image in the group analysis.

16.16 Realignment

16.16.1 1. What is realignment / motion correction?

In a perfect world, subjects would lie perfectly still in the scanner while you experimented on them. But, of course, in a perfect world, I'd be six foot ten with a killer Jump Hook, and my car would have those hubcaps that spin independently of the wheels. Sadly, I only have three of my original Camry hubcaps, and subjects are too darned alive to hold perfectly still. When your subject moves in the scanner, no matter how much, a couple things happen:

- Your voxels don't move with them. So the center of your voxel 10, 10, 10, say, used to be just on the front edge of your subject's brain, but now they've moved a millimeter backwards - and that same voxel now sampling from outside the brain. If you don't correct for that, you're going to get an blurry-looking brain when you average your functional effect over time. Think of the scanner as a camera taking a really long exposure - if your subject moves during the exposure, she'll look blurry.
- Their movement generates tiny inhomogeneities in the magnetic field. You've carefully prepared your magnetic field to be perfectly smooth everywhere in the scanner - so long as the subject is in a certain position. When she moves, she generates tiny "ripples" in the field that can cause transient artifacts. She also subtly changes the magnetic field for the rest of the experiment; if she gradually moves forward, for example, you may see the

back of her brain get gradually brighter over the experiment as she changes the field and moves through it. If you don't account for that, it'll look like the back of her brain is getting gradually more and more active.

- Realignment (also called motion correction - they're the same thing) mainly aims to help correct the first of these problems. Motion correction algorithms look across your images and try to "line up" each functional image with the one before it, so your voxels always sample the same location and you don't get blurring. The second problem is a little trickier - see the question below on including movement parameters in your design matrix. This issue also comes up in correcting for physiological movement, so check out *Physiology and fMRI* as well.

16.16.2 2. How do the major programs' algorithms work? How do they perform relative to each other?

SPM, AFNI, BrainVoyager, AIR 3.0, and most other major programs, all essentially use modifications of the same algorithm, which is the minimization of a least-squares cost function. The algorithm attempts to find the rigid-body movement parameters for each image that minimizes the voxel-by-voxel intensity difference from the reference image.

The particular implementation of the algorithm varies widely between programs, though. Ardekani et. al (below) does a detailed performance breakdown of SPM99 vs. AFNI98 vs. AIR 3.0 vs. TRU. AFNI is by far the fastest and also the most accurate at lower SNRs; SPM99, though slower, is the most accurate at higher SNRs. See below for more detail...

16.16.3 3. How much movement is too much movement?

Tough to give an exact answer, but Ardekani et. al find that SPM and AFNI can handle up to 10mm initial misalignment (summed across x/y/z dimensions) without significant trouble. Movement in any single dimension greater than the size of a single voxel between to images is probably worth looking at, and several images with greater than one-voxel motion in one run is a good guideline for concern.

16.16.4 4. How should you correct motion outliers? When should you just throw them out?

Attempting to correct extreme outliers is a tricky process. On the one hand, images with extremely distorted intensities due to motion can have a measurable distortion effect on your results. On the other hand, distinguishing intensity changes due to motion as opposed to task-related signal is by no means an easy process, and removing task-related signal can also measurably worsen your results (relative to both Type I and II errors).

Our current thinking in the lab is that outlier correction should be attempted only when you can find isolated scans who show significantly distorted global intensities (several standard deviations away from the mean) that are with a TR or two of a significant head movement. A significant head movement without a global intensity change is probably handled best by the realignment process itself; a significant intensity change without head motion may have nothing to do with motion.

Another option is to simply censor (i.e., not use) the images identified as iffy; this is easier in AFNI than in SPM. This has the disadvantage of possibly distorting your trial balancing in a given session if whole trials are removed, as well losing whatever task signal there may be in that scan. It has the advantage of being more statistically valid - outlier correction with interpolation obviously introduces new temporal correlation into your timeseries.

Several things might make a particular session entirely unusable: several isolated scans with head motion of greater than 10mm (summed across x/y/z); several scans with head motion in a single direction greater than the size of a single voxels; a run of several scans in a row with significant motion and significant intensity change; high correlation of your motion parameters with your task (see below). All subjects should be vetted for these problems before their results are taken seriously...

16.16.5 5. How can you tell if it's working? Not working?

Realignment in general is pretty robust; the least-squares algorithm will always produce some solution. It may, however, get caught in a non-optimal solution, particularly with scans that have a lot of motion and/or a big intensity change from the one before. It's difficult to evaluate realignment's effects post hoc just by looking at your results; the best way to make sure it's worked is visual inspection. SPM's "Check Reg" button will allow you to simultaneously display up to 15 images at once, side-by-side with a crosshair placed identically in each of them, to make sure a given voxel location lines up among several images. You may want to look particularly at scans you've identified with significant head motion, as well as comparing the first and last images in your run...

16.16.6 6. Should I include movement parameters in my design matrix? Why or why not?

In a nutshell, even after realignment, various effects like interpolation errors, changes in the shim from head movement, spin history effects, etc. can induce motion-correlated intensity changes in your data. Including your motion parameters in your design matrix can do a very good job of removing these changes; including values derived from these parameters, such as their temporal derivative, or sines of their values (see Grootonk et. al) can do an even better job.

The reason not to include these parameters is that there's a pretty good chance you also have actual task-related signal that happens to correlate with your motion parameters, and so removing all intensity changes correlated with motion may also significantly decrease your sensitivity to task-related signal. Before including these parameters in your matrix, you're probably wise to check how much your motion correlates with your task to make sure you're not inadvertantly removing the signal you want to detect.

16.16.7 7. What is 'unwarping'? Why is it included as a realignment option in SPM2? And when should I use it?

Head motion can cause artifacts for a variety of reasons - gradual changes in the shim, changes in dropout, changes in slice interpolations, spin-history effects - but certainly one of the big ones is motion-by-susceptibility interactions. In areas typically vulnerable to susceptibility-induced artifacts - artifacts caused by magnetic field distortion due to air-tissue interfaces - head motion can cause majors changes in those susceptibility regions, and intensities around the interface can change in unpredictable ways as the head moves. The guys at SPM describe it as being like a funhouse mirror near the interface - there's usually some spot of blackout (signal dropout) where susceptibility is really bad, but right around it, the image is distorted in weird ways, and sliding around can change those distortions in unpredictable ways.

Motion-by-susceptbility interaction is certainly one of the biggest sources of motion-related artifact, and some people think it's THE biggest, and so the "unwarp" utility in SPM2 is an attempt to specifically address it. Even if you get the head lined up image-to-image (realignment), these effects will remain, and so you can try and remove them additionally (unwarping). This is essentially a slightly less powerful but very much more specific version of including your motion parameters in the design matrix - you'll avoid almost all the problems about task-correlated motion that go with including your motion parameters as correlates, but (hopefully) you'll get almost all the same good effects. The benefits will be particularly noticeable in high-susceptibility regions (amygdala, OFC, MTL).

One BIG caveat about unwarping, though - as it's currently implemented, I believe it's ONLY usable for EPI images, NOT for spiral. So if you use spiral images, you shouldn't use this. But if you use EPI, it can be worth a try, particularly if you're looking at high-susceptibility regions.

16.16.8 8. What's the best base image to realign to? Is there any difference between realigning to an image at the beginning of the run and one in the middle of the run?

Not a huge difference, if any. AFNI has a program (findmindiff, I think) that identifies the image in a particular series that has the least difference from all the other images, which would be the ideal one to use. In practice, though, there's probably no significant difference between using that and simply realigning to the first image of the run, unless you have very large (10mm+) movement over the course of the run, in which case the session is probably of questionable use as well...

16.16.9 9. When is realigning a bad idea?

The trouble with the least-squares algorithm that realignment programs use is that it's easily fooled into thinking differences in intensity between images are all due to motion. If those differences are due to something else - task-related signal, or sudden global intensity changes - the realignment procedure can be fooled and come up with a bad realignment. If the realignment is particularly bad, it can completely obscure your signal, or (arguably worse) generate false activations! This is most pressing in the case of task-correlated motion (see below for discussion), but if you have significant global intensity shifts during your session that aren't motion-related, your realignment will probably introduce - rather than remove - error into your experiment. There are other realignment methods you can use to get around this, but they're slow. See Friere & Mangin, and the *Coregistration* page.

16.16.10 10. What can I do about task-correlated motion? What's the problem with it?

See Bullmore et. al, Field et. al, and Friere & Mangin for more details about this issue. The basic problem stems from the fact that head motion doesn't just rotate and shift the head in an intensity-invariant fashion. Head motion actually changes the image intensities, due to inhomogeneity in the magnetic field, changes in susceptibility, spin history effects, etc. If your subject's head motions are highly correlated with your task onsets or offsets, it can be impossible to how much of a given intensity change is due to head motion and how much is due to actual brain activation. The effect is that task-correlated motion can induce signficant false activations in your data. Including your motion parameters in your design matrix in this case, to try and account for these intensity changes, will hurt you the other way - you'll end up removing task-correlated signal as well as motion and miss real activations.

The extent of the problem can be significant. Field et. al, using a physical phantom (which doesn't have brain activations) were able to generate realistic-looking clusters of "activation," sometimes of 100+ voxels, with head movements of less than 1mm, simply by making the phantom movements increasingly correlated with their task design. Bullmore et. al point out that patient groups frequently have significant differences in how much task-correlated motion they have relative to normals, which can significantly bias between-group comparisons of activation.

Even worse, the least-squares algorithm commonly used to realign function images is biased by brain activations, because it assumes intensity changes are due to motion and attempts to correct for them. As Friere & Mangin point out, even if there's no motion at all, the presence of significant activations could cause the realignment to report motion parameters that have task-correlated motion!

So what can you do? First and foremost, you should always evaluate how correlated your subjects' motion is with your task - the parameters themselves and linear combinations of them. (An F-test can do this - we'll have a script available for this in the lab shortly.) The correlation of your parameters with your task is hugely more important than the size of your motion in generating false activations. Field demonstrated false activation clusters with correlations above r = 0.52. If your subject exhibits very high correlation, there's not much you can do - they're probably not usable, or at least their results should be taken with a grain of salt. There are some techniques (see Birn et. al, below) that may help distinguish activations from real motion, but they're not perfect...

Bullmore et. al, below, report some ways to account for task-correlated motion that may be useful.

Even without any task-correlated motion, though, you should be aware your motion parameters may be biased, as above, towards reporting a correlation. This is not usually a problem with relatively small activations; it may be bigger with very large signal changes. You can avoid the problem entirely by using a different realignment algorithm - based on mutual information, like the algorithms here (*Coregistration*) - but these are impossibly slow, and not practically usable.

Among the usable algorithms, Morgan et. al reported SPM99 was the best at avoiding false-positive voxels... Keep an eye out for more robust algorithms coming in the future, though... And you may want to try and use one of the prospective motion correction algorithms, as described in Ward et. al. at.

16.17 Scanning

This section is intended to address design-related questions that focus primarily on technical aspects about the scanner - things like TR, pulse sequence, slice thickness, etc. Obviously, setting your scanner parameters is mixed in heavily with your experimental design, so be sure to check out some other design-related pages:

- *Design*
- *Jitter*
- *Physiology and fMRI*

16.17.1 1. What pulse sequence shoudl I use (EPI or spiral)?: What are pros and cons of each? What do each of them get you?

- EPI: More widely used, and hence supported by all fMRI analysis programs. Some programs (FSL, or SPM's unwarping module, for example) do not support spiral data. Can be subject to less drop-out in some regions than spiral-in or spiral-out data alone. Can be easier to figure out what the slice ordering is.
- Spiral: Properly weighted and combined, spiral in-out shows significantly less signal drop-out and shows significantly greater activations in many areas of the brain, including ventromedial PFC, medial temporal lobe, etc. Effect is even more pronounced at higher field strengths (see Preston et. al). However, it is less widely supported, and Gary's trademark spiral i-o sequence may not even be physically possible on some other institutions' equipment.

16.17.2 2. What should your TR be? What are the tradeoffs, and what's the best tradeoff of coverage vs. speed for different types of analysis?

Bottom line: TR should be as short as possible, given how many slices you want to cover and the limits of your task. Gary's handout and monograph speak best to this issue, and are good quick reads. Decreasing your TR decreases your signal-to-noise ratio (SNR) in any one functional image, but because you have more images to work with, your overall SNR increases with decreasing SNR. Your TR, however, is limited by how many slices you want to take. On the 3T scanner here at Stanford, using spiral in-out, each slice takes approximately 65 msec/slice (TE = 30 msec), so you can get 15 full slices in 1 second (on the 1.5T, slices take about 75 msec, so you can get 13 full slices in 1 sec.). Your tradeoff is that with fewer slices, you have to either accept less coverage of the brain, or thicker slices, which will have poorer resolution in the z direction.

For certain experiments, then - ones focused on primary sensory on motor cortices, when you don't care about the rest of the brain - you can buy yourself shorter TRs by decreasing your number of slices, or you can increase your number of slices and make them smaller, while keeping your TR constant. Assuming you need full coverage of the brain, you can only decrease your TR by making your slices thicker, which you should do as much as possible within the constraints of your desired resolution.

Alternatively, in experimental designs where you're not particularly focused on timecourse information and where you already have good statistical power - namely, block-design experiments - you may want to get better resolution by increasing your number of slices and hence your TR. In event-related experiments, having a short TR becomes even more imporant, due to the relative lack of experimental power in such designs and relative importance of timecourse information.

16.17.3 3. What should your slice thickness be?

As thick as you think you can get away with. Increasing your slice thickness allows you to decrease your TR and maintain the same coverage, which is desirable as you get better SNR with decreasing TR. Alternatively, if you need good resolution in all dimensions, you can shrink your slice thickness at the expense of either brain coverage or having a longer TR.

16.17.4 4. What should your slice resolution / voxel size be?

64 x 64 is standard around here for full-brain coverage. With experiments focusing on smaller areas - primary motor and/or sensory cortices - something else (like 128 x 128) may be useful to get better resolution in a smaller area.

This differs from what size you interpolate your voxels to in normalization, which is covered in *Normalization*...

16.17.5 5. Should you acquire axially/coronally/something else? How come?

Big issue here, as I understand it, is that your slices are often thicker than your in-slice voxels, and hence your resolution is often poorest in the direction perpendicular to your slices. (Hence, if you acquire axially, your inferior-superior or z-direction resolution may not be great.) If you have a particular structure of interest, depending on its orientation, you may want to arrange your slices so as to get good resolution in the direction necessary to nail down that structure. Anyone else have any comments on this one?

16.17.6 6. BOLD vs. perfusion: what are pros and cons of each? What sorts of experiments would you use perfusion for?

Perfusion imaging - in which arterial blood is magnetically 'labeled' with an RF pulse, and then tracked as it moves through the brain - has two main advantages we discussed, one of which is thoroughly discussed in the Aguirre article. That advantage is the relatively different noise profile present within the perfusion signal. Unlike BOLD, perfusion noise doesn't have very much autocorrelation, which isn't by itself anything special, but means that perfusion contains much, much less noise relative to BOLD at very low experimental frequencies. There is more noise in general, though, in perfusion imaging, so in general SNR ratios are better for BOLD. But for experiments with very low task-switching frequency - say, blocks of 60s or more, even up to many minutes or hours - BOLD is almost useless, due to the preponderance of low-frequency noise, whereas the perfusion signal is unchanged. This means that with block lengths of longer than a minute, perfusion imaging is probably a better way to go, and experiment which previously weren't possible - block lengths of several minutes, or task switching taking place over several days - might be designed with perfusion.

Another feature of the noise in perfusion imaging is that it appears to be more reliable across subjects. While SNR within a given subject is higher for BOLD, group SNR appears (with limited data in Aguirre) to be higher in perfusion imaging. More research is needed on this subject, but this relative SNR advantage may be useful for experiments with small numbers of subjects, as across-subject variability is always the largest noise source for BOLD experiments, often by a huge factor.

The other primary advantage of perfusion relative to BOLD imaging is that the perfusion signal is an absolute number, rather than a contrast. Each voxel is given a physiologically intelligible value - amount of cerebral blood flow - which means that it can be especially useful for comparing groups of populations. Comparing the results of a particular contrast in depressed vs. normal subjects might not yield any results, for example, but overall blood flow might just be lower in the absolute in a particular regions for depressed subjects relative to normal subjects - which would be very interesting. The ability to compare baselines in perfusion is a strong case for using it in particular types of experiments where baseline information may be interesting.

16.18 Segmentation

16.18.1 1. What is segmentation?

Segmentation is the process by which you separate your brain pictures into different tissue types. You give the segmentation program a brain image, and it classifies every voxel by tissue type - grey matter, white matter, CSF, skull, etc. Some segmentation algorithms operate on a probabilistic basis rather than a "hard" classification (so one voxel might by 60% likely to be grey, 10% likely to be white, etc.). Some segmentation algorithms go further than tissue type, and classify individual anatomical regions as well. Segmentation algorithms often give back output images, consisting of all the grey voxels in the brain, for example.

A subset of segmentation algorithms focus only on the problem of separating brain from skull tissue; these are often called "skull-stripping" or "brain-extraction" algorithms. The problem of classifying brain from skull is slightly easier than classifying different brain tissue types, but many of the same problems are faced, so we lump them in together with general segmentation algorithms.

16.18.2 2. Why should you segment?

Lotta reasons. Might be you're interested in the details of the segmentation - how much gray matter is in a particular region, how much white matter, etc. A lot of those analyses fall under the label of voxel-based morphometry (VBM), discussed below in the Ashburner & Friston paper. Alternatively, you might be interested in masking your analysis with one of the segments and only examining activated voxels that are in gray matter in a particular region. You might want to segment only to increase the accuracy of another preprocessing step - you might care that your normalization, say, is especially good in gray matter while you don't care as much about its accuracy in white matter. Simply extracting the brain has even more utility; some analysis programs or preprocessing steps require you to strip skull tissue off the brain before using them. You might simply want to create an analysis mask of all the brain voxels and ignore the other ones.

All of those issues would require you to identify which voxels of your image (almost always anatomical) are gray matter, which are white, and which are CSF or skull or other stuff (or at least which are brain and which are not). You can do this by hand, but it's an arduous and hugely time-consuming process, infeasible for large groups of subjects. Several automated methods are available, though, to do it. Generally, the algorithms take some input image and produce labels for every voxel, assigning them to one of the categories above, or sometimes anatomical labels as well (see below). Alternativey, some algorithms exist that do a "soft" classification and assign each voxel a certain probability of being a particular tissue type. Which you use will depend on exactly what your goals for segmentation are.

Finally, segmentation algorithms are increasingly being used not only to separate tissue types but to automate the production of individualized anatomical ROIs. Would you like to hand-draw your caudate or thalamus, say, but figure it'll take too long or be too hard? Automated segmentation algorithms could be used to simplify the process.

16.18.3 3. What are the problems I might face with segmentation?

Segmentation algorithms face two big issues: intensity overlap and partial voluming. Intensity overlap refers to the fact that the intensity distributions for different tissue types aren't completely separate - they have significant overlap, such that a bright voxel might be a particularly bright gray matter voxel or, just as easily, a particularly bright white matter voxel. Because all segmentation algorithms have to work with is the intensity value at each voxel (and those around it), this poses a problem for hard classifications. As well, inhomogeneities in the magnetic field, susceptibility-induced magnetic changes or head motion during acquisition can all produce gradual shadings of light or dark in images that can make the different tissue types even harder to distinguish - a particular brightness level might be gray matter at the front of the head, but white matter at the back of the head. One way to address these problems is to take spatial location into account; at the simplest level, voxels can always be assigned a high probability of being the same tissue type as the voxels around them (spatial coherence), or one can use a more sophisticated method like incorporating a full prior probability atlas (see Fischl et. al and Marroquin et. al, below).

Partial voluming refers to the fact that even high-res MRI has a limited spatial resolution, and a given voxel might include signal from several different tissue types to varying degree. This is particularly important along tissue-type borders, where if an algorithm is biased towards one tissue type or another, estimates of one tissue type's volume within an area can be significantly inflated or deflated from reality. One way to address this problem is with "soft" classification - instead of semi-arbitarily assigning voxels to definite tissue types, one can assign voxels probabilities of being in a tissue type, and take that confidence level into account when deciding tissue volume, etc.

16.18.4 4. How are coregistration and segmentation related?

Fischl et. al make the point that the two processes operate on different sides of the same coin - each one can solve the other. With a perfect coregistration algorithm, you could be maximally confident that you could line up a huge number of brains and create a perfect probability atlas - allowing you the best possible prior probabilities with which to do your segmentation. In order to do a good segmentation, then, you need a good coregistration. But if you had a perfect segmentation, you could vastly improve your coregistration algorithm, because you could coregister each tissue type separately and greatly improve the sharpness of the edges of your image, which increases mutual information.

Fortunately, MI thus far appears to do a pretty good job with coregistration even in unsegmented images, breaking us out of a chicken-and-egg loop. But future research on each of these processes will probably include, to a greater and greater extent, the other process as well.

Check out *Coregistration* for more info on coregistration...

16.19 Slice Timing

16.19.1 1. What is slice timing correction? What's the point?

In multi-shot EPI (or spiral methods, which mimic them on this front), slices of your functional images are acquired throughout your TR. You therefore sample the BOLD signal at different layers of the brain at different time points. But you'd really like to have the signal for the whole brain from the same time point. If a given region that spans two slices, for example, all activates at once, you want to see what the signal looks like from the whole region at once; without correcting for slice timing, you might think the part of the region that was sampled later was more active than the part sampled earlier, when in fact you just sampled from the later one at a point closer to the peak of its HRF.

What slice-timing correction does is, for each voxel, examine the timecourse and shift it by a small amount, interpolating between the points you ACTUALLY sampled to give you back the timecourse you WOULD have gotten had you sampled every voxel at exactly the same time. That way you can make the assumption, in your modeling, that every point in a given functional image is the actual signal from the same point in time.

16.19.2 2. How does it work?

The standard algorithm for slice timing correction uses sinc interpolation between time points, which is accomplished by a Fourier transform of the signal at each voxel. The Fourier transform renders any signal as the sum of some collection of scaled and phase-shifted sine waves; once you have the signal in that form, you can simply shift all the sines on a given slice of the brain forward or backward by the appropriate amount to get the appropriate interpolation.

There are a couple pitfalls to this technique, mainly around the beginning and end of your run, highlighted in Calhoun et. al below, but these have largely been accounted for in the currently available modules for slice timing correction in the major programs.

16.19.3 3. Are there different methods or alternatives and how well do they work?

One alternative to doing slice-timing correction, detailed below in Henson et. al, is simply to model your data with an HRF that accounts for significant variability in when your HRFs onset - i.e., including regressors in your model that convolve your data with both a canonical HRF and with its first temporal derivative, which is accomplished with the 'hrf + temporal derivative' option in SPM. In terms of detecting sheer activation, this seems to be effective, despite the loss of some degrees of freedom in your model; however, your efficiency in estimating your HRF is very significantly reduced by this method, so if you're interested in early vs. late comparisons or timecourse data, this method isn't particularly useful.

Another option might be to include slice-specific regressors in your model, but I don't know of any program that currently implements this option, or any papers than report on it...

16.19.4 4. When should you use it?

Slice timing correction is primarily important in event-related experiments, and especially if you're interested in doing any kind of timecourse analysis, or any type of 'early-onset vs. late-onset' comparison. In event-related experiments, however, it's very important; Henson et. al show that aligning your model's timescale to the top or bottom slice can results in completely missing large clusters on the slice opposite to the reference slice without doing slice timing correction. This problem is magnified if you're doing interleaved EPI; any sequence that places adjacent slices at distant temporal points will be especially affected by this issue. Any event-related experiment should probably use it.

16.19.5 5. When is it a bad idea?

It's never that bad an idea, but because the most your signal could be distorted is by one TR, this type of correction isn't as important in block designs. Blocks last for many TRs and figuring out what's happening at any given single TR is not generally a priority, and although the interpolation errors introduced by slice timing correction are generally small, if they're not needed, there's not necessarily a point to introducing them. But if you're interested in doing any sort of timecourse analysis (or if you're using interleaved EPI), it's probably worthwhile.

16.19.6 6. How do you know if it's working?

Henson et. al and both Van de Moortele papers below have images of slice-time-corrected vs. un-slice-time-corrected data, and they demonstrate signatures you might look for in your data. Main characteristics might be the absence of significant differences between adjacent slices. I hope to post some pictures here in the next couple weeks of the SPM sample data, analyzed with and without slice timing correction, to explore in a more obvious way.

16.19.7 7. At what point in the processing stream should you use it?

This is the great open question about slice timing, and it's not super-answerable. Both SPM and AFNI recommend you do it before doing realignment/motion correction, but it's not entirely clear why. The issue is this:

- If you do slice timing correction before realignment, you might look down your non-realigned timecourse for a given voxel on the border of gray matter and CSF, say, and see one TR where the head moved and the voxel sampled from CSF instead of gray. This would results in an interpolation error for that voxel, as it would attempt to interpolate part of that big giant signal into the previous voxel.

- On the other hand, if you do realignment before slice timing correction, you might shift a voxel or a set of voxels onto a different slice, and then you'd apply the wrong amount of slice timing correction to them when you corrected - you'd be shifting the signal as if it had come from slice 20, say, when it actually came from slice 19, and shouldn't be shifted as much. There's no way to avoid all the error (short of doing a four-dimensional realignment process combining spatial and temporal correction - possibly coming soon), but I believe the current thinking is that doing slice timing first minimizes your possible error. The set of voxels subject to such an interpolation error is small, and the interpolation into another TR will also be small and will only affect a few TRs in the timecourse. By contrast, if one realigns first, many voxels in a slice could be affected at once, and their whole timecourses will be affected. I think that's why it makes sense to do slice timing first. That said, here's some articles from the SPM e-mail list that comment helpfully on this subject both ways, and there are even more if you do a search for "slice timing AND before" in the archives of the list.

16.19.8 8. How should you choose your reference slice?

You can choose to temporally align your slices to any slice you've taken, but keep in mind that the further away from the reference slice a given slice is, the more it's being interpolated. Any interpolation generates some small error, so the further away the slice, the more error there will be. For this reason, many people recommend using the middle slice of the brain as a reference, minimizing the possible distance away from the reference for any slice in the brain. If you have a structure you're interested in a priori, though - hippocampus, say - it may be wise to choose a slice close to that structure, to minimize what small interpolation errors may crop up.

16.19.9 9. Is there some systematic bias for slices far away from your reference slice, because they're always being sampled at a different point in their HRF than your reference slice is?

That's basically the issue of interpolation error - the further away from your reference slice you are, the more error you're going to have in your interpolation - because your look at the "right" timepoint is a lot blurrier. If you never sample the slice at the top of the head at the peak of the HRF, the interpolation can't be perfect there if you're interpolating to a time when the HRF should be peaking - but hopefully you have enough information about your HRF in your experiment to get a good estimation from other voxels. It's another argument for choosing the middle slice in your image - you want to get as much brain as possible in an area of low interpolation error (close to the reference slice).

16.19.10 10. How can you be sure you're not introducing more noise with interpolation errors than you're taking out with the correction?

Pretty good question. I don't know enough about signal processing and interpolation to say exactly how big the interpolation errors are, but the empirical studies below seem to show a significant benefit in detection by doing correction without adding much noise or many false positive voxels. Anyone have any other comments about this?

16.20 Smoothing

16.20.1 1. What is smoothing?

"Smoothing" is generally used to describe spatial smoothing in neuroimaging, and that's a nice euphamism for "blurring." Spatial smoothing consists of applying a small blurring kernel across your image, to average part of the intensities from neighboring voxels together. The effect is to blur the image somewhat and make it smoother - softening the hard edges, lowering the overall spatial frequency, and hopefully improving your signal-to-noise ratio.

16.20.2 2. What's the point of smoothing?

Improving your signal to noise ratio. That's it, in a nutshell. This happens on a couple of levels, both the single-subject and the group.

At the single-subject level: fMRI data has a lot of noise in it, but studies have shown that most of the spatial noise is (mostly) Gaussian - it's essentially random, essentially independent from voxel to voxel, and roughly centered around zero. If that's true, then if we average our intensity across several voxels, our noise will tend to average to zero, whereas our signal (which is some non-zero number) will tend to average to something non-zero, and presto! We've decreased our noise while not decreasing our signal, and our SNR is better. (Desmond & Glover demonstrate this effect with real data.)

At the group level: Anatomy is highly variable between individuals, and so is exact functional placement within that anatomy. Even with normalized data, there'll be some good chunk of variability between subjects as to where a given functional cluster might be. Smoothing will blur those clusters and thus maximize the overlap between subjects for a given cluster, which increases our odds of detecting that functional cluster at the group level and increasing our sensitivity.

Finally, a slight technical note for SPM: Gaussian field theory, by which SPM does p-corrections, is based on how smooth your data is - the more spatial correlation in the data, the better your corrected p-values will look, because there's fewer degree of freedom in the data. So in SPM, smoothing will give you a direct bump in p-values - but this is not a "real" increase in sensitivity as such.

16.20.3 3. When should you smooth? When should you not?

Smoothing is a good idea if:

- You're not particularly concerned with voxel-by-voxel resolution.
- You're not particularly concerned with finding small (less than a handful of voxels) clusters.
- You want (or need) to improve your signal-to-noise ratio.
- You're averaging results over a group, in a brain region where functional anatomy and organization isn't precisely known.
- You're using SPM, and you want to use p-values corrected with Gaussian field theory (as opposed to FDR).

Smoothing'd not a good idea if:

- You need voxel-by-voxel resolution.
- You believe your activations of interest will only be a few voxels large.
- You're confident your task will generate large amounts of signal relative to noise.
- You're working primarily with single-subject results.
- You're mainly interested in getting region-of-interest data from very specific structures that you've drawn with high resolution on single subjects.

16.20.4 4. At what point in your analysis stream should you smooth?

The first point at which it's obvious to smooth is as the last spatial preprocessing step for your raw images; smoothing before then will only reduce the accuracy of the earlier preprocessing (normalization, realignment, etc.) - those programs that need smooth images do their own smoothing in memory as part of the calculation, and don't save the smoothed versions. One could also avoid smoothing the raw images entirely and instead smooth the beta and/or contrast images. In terms of efficiency, there's not much difference - smoothing even hundreds of raw images is a very fast process. So the question is one of performance - which is better for your sensitivity?

Skudlarski et. al evaluated this for single-subject data and found almost no difference between the two methods. They did find that multifiltering (see below) had greater benefits when the smoothing was done on the raw images as opposed to the statistical maps. Certainly if you want to use p-values corrected with Gaussian field theory (a la SPM), you need to smooth before estimating your results. It’s a bit of a toss-up, though...

16.20.5 5. How do you determine the size of your kernel? Based on your resolution? Or structure size?

A little of both, it seems. The matched filter theorem, from the signal processing field, tells us that if we’re trying to recover a signal (like an activation) in noisy data (like fMRI), we can best do it by smoothing our data with a kernel that’s about the same size as our activation.

Trouble is, though, most of us don’t know how big our activations are going to be before we run our experiment. Even if you have a particular structure of interest (say, the hippocampus), you may not get activation over the whole region - only a part.

Given that ambiguity, Skudlarski et. al introduce a method called multifiltering, in which you calculate results once from smoothed images, and then a second set of results from unsmoothed images. Finally, you average together the beta/con images from both sets of results to create a final set of results. The idea is that the smoothed set of results preferentially highlight larger activations, while the unsmoothed set of results preserve small activations, and the final set has some of the advantages of both. Their evaluations showed multifiltering didn’t detect larger activations (clusters with radii of 3-4 voxels or greater) as well as purely smoothed results (as you might predict) but that over several cluster sizes, multifiltering outperformed traditional smoothing techniques. Its use in your experiment depends on how important you consider detecting activations of small size (less than 3-voxel radius, or about).

Overall, Skudlarski et. al found that over several cluster sizes, a kernel size of 1-2 voxels (3-6 mm, in their case) was most sensitive in general.

A good rule of thumb is to avoid using a kernel that’s significantly larger than any structure you have a particular a priori interest in, and carefully consider what your particular comfort level is with smaller activations. A 2-voxel-radius cluster is around 30 voxels and change (and multifiltering would be more sensitive to that size); a 3-voxel-radius cluster is 110 voxels or so (if I’m doing my math right). 6mm is a good place to start. If you’re particularly interested in smaller activations, 2-4mm might be better. If you know you won’t care about small activations and really will only look at large clusters, 8-10mm is a good range.

16.20.6 6. Should you use a different kernel for different parts of the brain?

It’s an interesting question. Hopfinger et. al find that a 6mm kernel works best for the data they examine in the cortex, but a larger kernel (10mm) works best in subcortical regions. This might be counterintuitive, considering the subcortical structures they examine are small in general than large cortical activations - but they unfortunately don’t include information about the size of their activation clusters, so the results are difficult to interpret. You might think a smaller kernel in subcortical regions would be better, due to the smaller size of the structures.

Trouble is, figuring out exactly which parts of the brain to use a different size of kernel on presupposes a lot of information - about activation size, about shape of HRF in one region vs. another - that pretty much doesn’t exist for most experimental set-ups or subjects. I would tend to suggest that varying the size of the kernel for different regions is probably more trouble than it’s worth at this point, but that may change as more studies come out about HRFs in different regions and individualized effects of smoothing. See Kiebel and Friston, though, for some advanced work on changing the shape of the kernel in different regions...

16.20.7 7. What does it actually do to your activation data?

About what you’d expect - preferentially brings out larger activations. Check out White et. al for some detailed illustrations. We hope to have some empirical results and maybe some pictures up here in the next few weeks...

16.20.8 8. What does it do to ROI data?

Great question, and not one I've got a good answer for at the moment. One big part of the answer will depend on the ratio of your smoothing kernel size to your ROI size. Presumably, assuming your kernel size is smaller than your ROI, it may help improve SNR in your ROI, but if the kernel and ROI are similar sizes, smoothing may also blur the signal such that your structure contains less activation. With any luck, we can do a little empirical testing on this questions and have some results up here in the future...

16.21 SPM in a Nutshell

SPM is a software package designed to analyze brain imaging data from PET or fMRI and output a variety of statistical and numerical measures that tell you, the researcher, what parts of your subjects' brains were signficantly "activated" by different conditions of your experiment. There are a couple of phases of analyzing data with SPM: spatial preprocessing, model estimation, and results exploration. This page aims to give you a nutshell explanation of what's actually happening in each of those phases (particularly model estimation) and what some of the files floating around your results directories are for. Links to pages with more detail about each aspect of the analysis are down at the bottom of this page.

Spatial preprocessing is conceptually the most straightforward part of SPM analysis. During this phase, you can align your images with each other, warp them (normalize) so that each subject's anatomy is roughly the same shape, correct them for differences in slice time acquisition, and smooth them spatially.

These steps are used for a couple of reasons. Registration and normalization aim to line images from a single subject up (since subjects' heads move slightly during the experiment) and normalization aims to stretch and squeeze the shape of the images so that their anatomy roughly matches a standard template; both of these aim to make localizing your activations easier and more meaningful, by making individual voxels' locations in a given image file match up in a standard way to a particular anatomical location. Slice timing correction and smoothing both enable SPM to make certain assumptions about the data images - that each whole image occurred at a particular point in time (as opposed to slices being taken over the course of an image acquisition, or TR), and that noise in an image is distributed in a relatively random and independent fashion (as opposed to being localized).

Model estimation is the heart of the SPM program, and it's also the most conceptually complex. What you as a researcher want to know from your data is essentially: what (if any) parts of my subject's brain were brighter during one part of my experiment relative to another part? Another way of putting this question might be this: You as a researcher have a hypothesis or model of what happened in an experiment; you have a list of different conditions and when each of them took place, and your model of the person's brain is that there was some kind of reaction in the brain for every stimulus that happened. How good a fit, then, does your hypothetical model provide to the actual MRI data you saw from the person's brain? Specifically, are there particular locations in the brain where your model was a very good fit, and others where it wasn't a good fit? The main work SPM does is to try and find those locations, because locations where your hypothesis proves to be a good fit can be described as "responding" somehow to the conditions in your experiment.

When SPM estimates a model, what it's doing is essentially a huge multiple regression at each voxel of your subject's brain, to see how well the data across the experiment fits your hypothesis, which you describe to SPM as a design matrix. When you tell SPM what your conditions are, what your onset vectors are, etc., it sets up this matrix as a guess at what contribution each condition might make to every image in your experiment. As part of that guess, it automatically convolves the effects of the hemodynamic response function with your stimulus vectors, as well as doing some temporal filtering to make sure it ignores changes in the data that aren't relevant to your conditions.

Once the design matrix has been set up, SPM walks through each voxel in the brain, and does a multiple regression on the data at that point that estimates how much of a contribution every condition in your experiment made to the data and how much error was left over after all the conditions you specified are taken into account. The fit of this regression line is important - how much error is left at the voxel tells you how good your model was - but for most purposes, the actual slope of the regression line is more important. A large positive or negative partial regression slope for a given condition tells us that that condition had a large influence in determining the data at that voxel. This slope, called the beta weight or parameter weight for that condition, is saved by SPM in the beta_* images - one for each column of the design matrix, where each voxel gives the beta weight for that condition at that point.

After the model estimation is complete, you now have a set of data telling you how big an effect each condition of your experiment had at each voxel. By itself, this information may not be useful - in most fMRI experiments, any condition by itself accounts for a tiny portion of the variance. What is useful to know, though, is if one condition made a significantly greater contribution than another condition did. This is where results analysis, and specifically contrast analysis, comes in. When you evaluate your results, SPM asks you to specify a contrast in terms of weights for each conditions. If you have only two conditions in your experiment, assuming your design matrix was (A B), then a contrast vector of (1 -1) would tell SPM you wanted to see at which voxels A had a signficantly larger contribution to brain activity than B did. SPM takes this contrast vector and literally uses it to make a weighted sum of the beta images it's just created; this new image, created by giving each beta image the weight you specified and adding them together, is a con_* image. SPM then looks across the con image at the distribution of weighted parameter values, and combines them with its estimate of the leftover variance from the model estimation, and assigns every voxel a T-statistic (creating an spm_T* image). When you ask SPM to only show you the voxels that are significantly active at a certain p-threshold, it looks at that T-stat image and finds only the voxels whose t-statistics are so large as to fit above that probability threshold - voxels where their weighted parameter values were so large as to be statistically unlikely at your specified level of significance.

Those voxels are where brain activity in your experiment was heavily influenced by one condition in your experiment more than another condition, so heavily influenced as to make it unlikely that activity was just noise. Those voxels were brighter / more intense in one condition of your experiment than they were in another with great reliability, and so they're considered active.

For more detail about a particular aspect of spatial preprocessing, check out the individual FAQ pages at:

- *Coregistration*
- *Realignment*
- *Normalization*
- *Smoothing*
- *Slice Timing*

16.22 Temporal Filtering

16.22.1 1. Why do filtering? What's it buy you?

Filtering in time and/or space is a long-established method in any signal detection process to help "clean up" your signal. The idea is if your signal and noise are present at separable frequencies in the data, you can attenuate the noise frequencies and thus increase your signal to noise ratio.

One obvious way you might do this is by knocking out frequencies you know are too low to correspond to the signal you want - in other words, if you have an idea of how fast your signal might be oscillating, you can knock out noise that is oscillating much slower than that. In fMRI, noise like this can have a number of courses - slow "scanner drifts," where the mean of the data drifts up or down gradually over the course of the session, or physiological influences like changes in basal metabolism, or a number of other sources. This type of filtering is called "high-pass filtering," because we remove the very low frequencies and "pass through" the high frequencies. Doing this in the spatial domain would correspond to highlighting the edges of your image (preserving high-frequency information); in the temporal domain, it corresponds to "straightening out" any large bends or drifts in your timecourse. Removing linear drifts from a timecourse is the simplest possible high-pass filter.

Another obvious way you might do this would be the opposite - knock out the frequencies you know are too high to correspond to your signal. This removes noise that is oscillating much faster than your signal from the data. This type of filtering is called "low-pass filtering," because we remove the very high frequencies and "pass through" the low frequencies. Doing this in the spatial domain is simply spatial smoothing (see *Smoothing*); in the temporal domain, it corresponds to temporal smoothing. Low-pass filtering is much more controversial than high-pass filtering. Finally, you could apply combinations of these filters to try and restrict the signal you detect to a specific band of frequencies, preserving only oscillations faster than a certain speed and slower than a certain speed. This is called "band-pass filtering," because we "pass through" a band of frequencies and filter everything else out, and is usually implemented in neuroimaging as simply doing both high-pass and low-pass filtering separately.

In all of these cases, the goal of temporal filtering is the same: to apply our knowledge about what the BOLD signal "should" look like in the temporal domain in order to remove noise that is very unlikely to be part of the signal. This buys us better SNR, and a much better chance of detecting real activations and rejecting false ones.

16.22.2 2. What actually happens to my signal when I filter it? How about the design matrix?

Filtering is a pretty standard mathematical operation, so all the major neuroimaging programs essentially do it the same way. We'll use high-pass as an example, as low-pass is no longer standard in most neuroimaging programs. At some point before model estimation, the program will ask the user to specify a cutoff parameter in Hz or seconds for the filter. If specified in seconds, this cutoff is taken to mean the period of interest of the experiment; frequencies that repeat over a timescale longer than the specified cutoff parameter are removed. Once the design matrix is constructed but before model estimation has begun, the program will filter each voxel's timecourse (the filter is generally based on some discrete cosine matrix) before submitting it to the model estimation - usually a very quick process. A graphical representation of the timecourse would show a "straightening out" of the signal timecourse - oftentime timecourses will have gradual linear drifts or quadratic drifts, or even higher frequency but still gradual bends, which are all flattened away after the filtering.

Other, older methods for high-pass filtering simply included a set of increasing-frequency cosines in the design matrix (see Holmes et. al below), allowing them to "soak up" low-frequency variance, but this is generally not done explicitly any more.

Low-pass filtering proceeds much the same way, but the design matrix is also usually filtered to smooth out any high frequencies present in it, as the signal to be detected will no longer have them. Low-pass filters are less likely to be specified merely with a lower-bound period-of-interest cutoff; oftentimes low-pass filters are constructed deliberately to have the same shape as a canonical HRF, to help highlight signal with that shape (as per the matched-filter theorem).

16.22.3 3. What's good about high-pass filtering? Bad?

High-pass filtering is relatively uncontroversial, and is generally accepted as a good idea for neuroimaging data. One big reason for this is that the noise in fMRI isn't white - it's disproportionately present in the low frequencies. There are several sources for this noise (see *Physiology and fMRI* and *Basic Statistical Modeling* for discussions of some of them), and they're expressed in the timecourses sometimes as linear or higher-order drifts in the mean of the data, sometimes as slightly faster but still gradual oscillations (or both). What's good about high-pass filtering is that it's a straightforward operation that can attenuate that noise to a great degree. A number of the papers below study the efficacy of preprocessing steps, and generally it's found to significantly enhance one's ability to detect true activations.

The one downside of high-pass filtering is that it can sometimes be tricky to select exactly what one's period of interest is. If you only have a single trial type with some inter-trial interval, then your period of interest of obvious - the time from one trial's beginning to the next - but what if you have three or four? Or more than that? Is it still the time from one trial to the next? Or the time from one trial to the next trial of that same type? Or what? Skudlarski et. al point out that a badly chosen cutoff period can be significantly worse than the simplest possible temporal filtering, which would just be removing any linear drift from the data. If you try and detect an effect whose frequency is lower than your cutoff, the filter will probably knock it completely out, along with the noise. On the other hand, there's enough noise at low frequencies to almost guarantee that you wouldn't be able to detect most very slow anyways. Perfusion imaging does not suffer from this problem, one of its benefits - the noise spectrum for perfusion imaging appears to be quite flat.

16.22.4 4. What's good about low-pass filtering? Bad?

Low-pass filtering is much more controversial in MRI, and even in the face of mounting empirical evidence that it wasn't doing much good, the SPM group long offered some substantial and reasonable arguments in favor of it. The two big reasons offered in favor of low-pass filtering broke down as:

- The matched-filter theorem suggests filtering our timecourse with a filter shaped like an HRF should enhance signals of that shape relative to the noise, and
- We need to modify our general linear model to account for all the autocorrelation in fMRI noise; one way of doing that is by conditioning our data with a low-pass filter - essentially 'coloring' the noise spectrum, or introducing our own autocorrelations - and assuming that our introduced autocorrelation 'swamps' the existing autocorrelations, so that they can be ignored. (See *Basic Statistical Modeling* for more on this.) This was a way of getting around early ignorance about the shape of the noise spectrum in fMRI and avoiding the computational burden of approximating the autocorrelation function for each model. Even as those burdens began to be overcome, Friston et. al pointed out potential problems with pre-whitening the data as opposed to low-pass filtering, relating to potential biases of the analysis.

However, the mounting evidence demonstrating the failure of low-pass filtering, as well as advances in computation speed enabling better ways of dealing with autocorrelation, seem to have won the day. In practice, low-pass filtering seems to have the effect of greatly reducing one's sensitivity to detecting true activations without significantly enhancing the ability to reject false ones (see Skudlarksi et. al, Della-Maggiore et. al). The problem with low-pass filtering seems to be that because noise is not independent from timepoint to timepoint in fMRI, 'smoothing' the timecourse doesn't suppress the noise but can, in fact, enhance it relative to the signal - it amplifies the worst of the noise and smooths the peaks of the signal out. Simulations with white noise show significant benefits from low-pass filtering, but with real, correlated fMRI noise, the filtering because counter-effective. Due to these results and a better sense now of how to correctly pre-whiten the timeseries noise, low-pass filtering is now no longer available in SPM2, nor is it allowed by standard methods in AFNI or BrainVoyager.

16.22.5 5. How do you set your cutoff parameter?

Weeeeelll... this is one of those many messy little questions in fMRI that has been kind of arbitrarily hand-waved away, because there's not a good, simple answer for it. You'd to like to filter out as much noise as possible - particularly in the nasty part of the noise power spectrum where the noise power increases abruptly - without removing any important signal at all. But this can be a little trickier than it sounds. Roughly, a good rule of thumb might be to take the 'fundamental frequency' of your experiment - the time between one trial start and the next - and double or triple it, to make sure you don't filter out anything closer to your fundamental frequency.

SPM99 (and earlier) had a formula built in that would try and calculate this number. But if you had a lot of trial types, and some types weren't repeated for very long periods of time, you'd often get filter sizes that were way too long (letting in too much noise). So in SPM2 they scrapped the formula and now issue a default filter size of 128 seconds for everybody, which isn't really any better of a solution.

In general, default sizes of 100 or 128 seconds are pretty standard for most trial lengths (say, 8-45 seconds). If you have particularly short trials (less than 10 seconds) you could probably go shorter, maybe more like 60 or 48 seconds. But this is a pretty arbitrary part of the process. The upside is that it's hard to criticize an exact number that's in the right ballpark, so you probably won't get a paper rejected purely because your filter size was all wrong.

CHAPTER SEVENTEEN

GLOSSARY

This Glossary wasn't created by myself. Its content is almost exclusively from the Imaging Knowledge Base - Glossary (http://mindhive.mit.edu/node/71) from the Gabrieli Lab at MIT (http://gablab.mit.edu/). It is my thanks to them, for generating such an exhaustive Glossary.

Index: *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* | *U* | *V* | *W* | *X* | *Y* | *Z* | *Numbers*

Note: To help me to extend this glossary, please feel free to leave a comment at the bottom to recommend additions and to clear misapprehension.

17.1 A

17.1.1 AC-PC, AC-PC line

Stands for "anterior commissure-posterior commissure." Used to describe the hypothetical line between the anterior commissure (a frontal white matter tract used as the origin of the Talairach coordinate system) and the posterior commissure (another white matter tract in the midbrain). A brain which is properly aligned to Talairach space has the line between the AC and the PC as exactly horizontal.

17.1.2 affine

In fMRI, a term for certain types of spatial transformations which add together linearly. From Mathworld (http://mathworld.wolfram.com/AffineTransformation.html): "An affine transformation is any transformation that preserves collinearity (i.e., all points lying on a line initially still lie on a line after transformation) and ratios of distances (e.g., the midpoint of a line segment remains the midpoint after transformation)." This includes all translations, rotations, zooming, or shearing (think 'squeezing' one end of a square such that it becomes a trapezoid). Importantly, affine transformations affect the whole image; no affine transformation can tweak one local part of an image and leave the rest exactly the same. The first step in SPM's normalization process is affine, generally followed by nonlinear normalization.

17.1.3 anatomical ROI

A region of interest (ROI) in the brain that is constructed from anatomical data (as opposed to functional activation data). Any ROI that is an anatomical structure in the brain - the inferior frontal gyrus, the amygdala, the posterior half of BA 32 - is an anatomical ROI.

17.1.4 anisotropic

The opposite of *isotropic*. In other words, *not* the same size in all directions. Anisotropy (the degree of anisotropicness) is one measure used in *Diffusion Tensor Imaging (DTI)* to determine the direction of white-matter fibers. A *smoothing kernel* or *voxels* can also be anisotropic.

17.1.5 ANOVA

Stands for ANalysis Of VAriance. A standard statistical tool used to find differences between the distributions of several groups of numbers. Differs from simpler tests like the t-test in that it can test for differences among many groups, not just two groups. The standard ANOVA model is used in neuroimaging primarily at the group level, to test for differences between several groups of subjects. However, the ANOVA is essentially the same thing as an F-test (test of the F-statistic), which is often used at the individual level as well to test several linear constraints on a model simultaneously. Check out Random and Fixed Effects FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#random-and-fixed-effects) for more info on group testing, and Contrasts FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#contrasts) for more info on F-tests.

17.1.6 AR(1)

AR(1) or AR(1) + w (or (AR(2), AR(3), etc.): Terms used to describe different models of *autocorrelation* in your fMRI data. See *autocorrelation* below for more info. AR stands for autoregression. AR models are used to estimate to what extent the noise at each time point in your data is influenced by the noise in the time point (or points) before it. The amount of autocorrelation of noise is estimated as a model parameter, just like *beta weights*. The difference between AR(1), AR(2), AR(1) + w, etc., is in which parameters are estimated. An AR(1) model describes the autocorrelation function in your data by looking only at one time point before each moment. In other words, only the correlation of each time point to the first previous time point is considered. In an AR(2) model, the correlation of each time point to the first previous time point and the second previous time point is considered; in an AR(3) model, the three time points before each time point are considered as parameters, etc. The "w" in AR(1) + w stands for "white noise." An AR(1) + w model assumes the value of noise isn't solely a function of the previous noise; it also includes a random white noise parameter in the model. AR(1) + w models, which are used in SPM2 and other packages, seem to do a pretty good job describes the "actual" fMRI noise function. A good model can be used to remove the effects of noise correlation in your data, thus validating the assumptions of the general linear model. See Temporal Filtering FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#temporal-filtering) for more info.

17.1.7 artifact

Essentially any noise in the fMRI signal that's localized in either space or time is generally referred to as an artifact. Common artifacts are caused by head motion, physiological motion (cardiac, respiration, etc.), or problems in the scanner itself.

17.1.8 autocorrelation

One major problem in the statistical analysis of fMRI data is the shape of fMRI noise. Analysis with the *general linear model (GLM)* assumes each timepoint is an independent observation, implying the noise at each timepoint is independent of the noise at the next timepoint. But several empirical studies have shown that in fMRI, that assumption's simply not true. Instead, the amount of noise at each timepoint is heavily correlated with the amount of noise at the timepoints before and after. fMRI noise is heavily "autocorrelated," i.e., correlated with itself. This means that each timepoint isn't an independent observation - the temporal data is essentially heavily smoothed, which means any statistical analysis that assumes temporal independence will give biased results.

The way to deal with this problem is pretty well-established in other scientific domains. If you can estimate what the autocorrelation function is - in other words, what, exactly, is the degree of correlation of the noise from one timepoint to the next - than you can remove the amount of noise that is correlated from the signal, and hence render your noise "white," or random (rather than correlated). This strategy is called *pre-whitening*, and is referred to in some fMRI packages as autocorrelation correction. The models used to do this in fMRI are mostly *AR(1)* + w models, but sometimes more complicated ones are used. See Basic Statistical Modeling FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#basic-statistical-modeling) for more info on autocorrelation correction.

17.2 B

17.2.1 B-spline, B-spline interpolation

A type of spline which is the generalization of the Bezier curve. Don't know what I'm talking about? Neither do I. The nice folks at MathWorld - Wolfram (http://mathworld.wolfram.com/) have this to say about them: B-Spline (http://mathworld.wolfram.com/B-Spline.html). Essentially, though, a B-spline is a type of easily describable and computable function which can take many locally smooth but globally arbitrary shapes. This makes them very nice for interpolation. SPM2 has ditched sinc interpolation in all of its resampling/interpolation functions (like normalization or coregistration - anything involving resampling and/or reslicing). Instead, it's now using B-spline interpolation, improving both computational speed and accuracy.

17.2.2 band-pass filter

The combination of a *high-pass filter* and *low-pass filter*. Band-pass filters only allow through a certain "band" of frequencies, while attenuating or knocking out everything outside that band. A well-designed band-pass filter would be great for fMRI experiments, because fMRI experiments generally have most of their frequencies in a certain band that's separable from the frequencies of fMRI noise. So if you could focus a band-pass filter on your experimental frequencies, you could knock out almost all of your noise. In practice, though, it's tricky to design a really good band-pass filter, and since most of the noise in fMRI is low-frequency, using only a high-pass filter works almost as well as band-pass filtering.

17.2.3 baseline

1. The point from which deviations are measured. In a signal measure like % signal change, the baseline value is the answer to, "Percent signal change *from what?*" It's the zero point on a % signal change plot.
2. A condition in your experiment that's intended to contain all of the cognitive tasks of your experimental condition - except the task of interest. In fMRI, you generally can only measure differences between two conditions (not anything absolute about one condition). So an fMRI baseline task is one where the person is doing everything you're not interested in, and not doing the thing you're interested in. This way you can look at signal during the baseline, subtract it from signal during the experimental condition, and be left with only the signal from the task of interest. Designing a good baseline is crucially important to your experiment. Resting with the eyes open is a common baseline for certain types of experiment, but inappropriate for others, where cognitive activity during rest may corrupt your results. In order to get good estimates of the shape of your HRF, you need to have a baseline condition (as opposed to several experimental conditions). Check out Design FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#design) for more.

17.2.4 basis function

One way to look for fMRI activation in the brain is to assume you know the exact shape of the HRF, and look for signals that match that shape. This is the most common way to analyze fMRI data. It suffers, though, in the case where the

HRF may not be exactly the same shape from one subject, one region, or even one task, to the next - which we know is true to some degree. Another way is to assume you know nothing about the shape of the HRF and separately estimate its value at every timepoint at every voxel. This is a *FIR (Finite Impulse Response) model*, and it's more common these days. But it suffers because it gives up many degrees of freedom in order to estimate a ton of parameters. A third way is to assume you know *something* about the shape of the response - maybe something as simple as "it's periodic," or something as complicated as "it looks kind of like one of these three or four functions here." This is the basis function approach, and the basis functions are the things you think "look" kind of like the HRF you want to estimate. They could be sines or cosines of different periods, which assumes very little about the shape except its periodicity, or they could be very-HRF looking things like the temporal and dispersion derivatives of the HRF. The basis function approach is kind of a middle way between the standard analysis and the FIR model. You only estimate parameters for each of your basis functions, so you get more power than the FIR model. But you aren't assuming you know the exact shape of your HRF, so you get more efficiency and flexibility than the standard analysis. You allow the HRF to vary somewhat - within the space defined by your basis functions - from voxel to voxel or condition to condition, but you still bring some prior knowledge about the HRF to bear to help you. Check out Design FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#design) and HRF FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#hrf) for more info on the basis function approach.

17.2.5 batch, batch script

Analysis programs with graphical interfaces are nice. But sometimes you don't want to have to push sixteen buttons and type in fourteen options to have to analyze every individual subject in your experiment. It takes a bunch of your time, and you'll probably screw it up and have to start over at some point. So many programs - SPM, AFNI, BrainVoyager - offer a "batch mode," where you can enter in the options you'd like in some sort of scripting language and then just set it to run the program in an automated function, according to the instructions in your batch script.

17.2.6 beta images

Also called a parameter images. It's a voxel-by-voxel summary of the *beta weights* for a given condition. Usually it's written as an actual image file or sub-dataset, so you could look at it just like a regular brain image, exploring the beta weight at each voxel. In SPM, you get one of these written out for every column in your design matrix - one for each experimental effect for which you're estimating parameter values.

17.2.7 beta weights

Also called parameter weights, parameter values, etc. This is the value of the parameter estimated for a given effect / column in your design matrix. If you think of the general linear model as a multiple regression, the beta weight is the slope of the regression line for this effect. The parameter gets its name as a "beta" weight from the standard regression equation: Y = BX + E. Y is the signal, X is the design matrix, E is error, and B is a vector of beta weights, which estimate how much each column of the design matrix contributes to the signal. Beta weights can be examined, summed, and contrasted at the voxel-wise level for a standard analysis of fMRI results. They can also be aggregated across regions or correlated between subjects for a more region-of-interest-based analysis. Check out ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) for more info on beta weights and ROIs.

17.2.8 block design

A type of experiment in which different types of trials are not intermixed randomly, but rather happen in blocks. So you might have 30 seconds in a row of condition A, followed by 30 seconds of condition B, followed by 30 seconds of A again, etc. Used even with shorter trials - that 30 seconds might be looking at a single flashing checkerboard, or it might be six trials of faces to look at. Block designs were the earliest type of design for fMRI and PET, and remain among the simpler designs to analyze and interpret. They have very high power, because the summing of HRF responses across repeated trials means you can often get higher peaks of activation during a block than for an isolated shorter trials. They suffer from very low efficiency (ability to estimate the shape of the HRF).

17.2.9 BOLD (blood oxygen level-dependent) signal

This is the type of signal that is measured during an fMRI acquisitiom. Check out Wikipedia's fMRI page (https://en.wikipedia.org/wiki/Functional_magnetic_resonance_imaging) for a primer on fMRI signal, but the nutshell version is this: When neurons fire (or increase their firing rate), they use up oxygen and various nutrients. The brain's circulatory system responds by flooding the firing region with more highly-oxygenated blood than it needs. The effect is that the blood oxygen level in the activated region increases slightly. Oxygenated blood has a slightly different magnetic signature than de-oxygenated blood, due to the magnetic characteristics of hemoglobin. So with the right *pulse sequence*, an MRI scanner can detect this difference in blood oxygen level. The signal that is thus read in fMRI is called BOLD, or blood oxygen level-dependent. MRI can be used to measure other things in the brain as well - *perfusion* being among them - but BOLD signal is the primary foundation of most fMRI research. Check out Physiology and fMRI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#physiology-and-fmri) for more details.

17.2.10 bootstrapping

A statistics method used when you have to test a distribution without knowing much about its true underlying variance or mean or anything. The skeleton of the method is essentially to build up a picture of the possible space of the distribution by re-shuffling the elements it's made up of to form new, random distributions. Bootstrapping is widely used in many quantitative scientific domains, but it's only recently become of interest in neuroimaging analysis. Some papers have argued that under certain conditions, bootstrapping and other nonparametric ways of testing hypotheses make the most sense to test statistical hypotheses in fMRI. *Permutation test* is the neuroimaging concept most related to boostrapping, and it's explored in P threshold FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#p-threshold).

17.2.11 Brodmann areas

An area of the brain that is distinct at the cytoarchitectonic (cellular) level from those around it. There are 52 Brodmann areas, originally defined by Korbinian Brodmann. Many of them map onto various distinct anatomical structures, but many also simply subdivide larger gyri or sulci. Mark Dubin at the University of Colorado has a great map of the areas: Brodmann map (http://spot.colorado.edu/~dubin/talks/brodmann/brodmann.html). They are often used as *anatomical ROI*, but be careful: they have significant variability from person to person in location and function. It's not clear how well functional activation maps onto most Brodmann areas. See ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) for more.

17.3 C

17.3.1 canonical HRF

A model of an "average" HRF. Intended to describe the shape of a generic HRF; given this shape and the design matrix, an analysis package will look for signals in the fMRI data whose shape matches the canonical HRF. The different analysis packages (SPM, AFNI, BrainVoyager, etc.) use slightly different canonical HRFs, but they all share the same basic features - a gradual rise up to a peak around six seconds, followed by a more gradual fall back to baseline. Some progams model a slight undershoot; some don't. See HRF FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#hrf) for more.

17.3.2 chronometry

A technique in psychology in which the experimenter tries to figure out something about the processes underlying a task by the time taken to do the task and various portions of it. Some of the original chronometric experiments were done with reaction times, having subjects do various stages of an experiment to see whether some parameter might vary the reaction time for one stage and not another. Chronometric experiments have just started cropping up in fMRI. They attempt to determine not just the location of activations, but their sequence as well. This is generally done by getting an extremely accurate estimate of the shape of the HRF and exactly when it begins during the task. See Mental Chronometry FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#mental-chronometry) for more.

17.3.3 cluster

A group of active voxels that are all adjacent, without any breaks. Clusters may include holes, but there has to be a contiguous link (vertical, horizontal or diagonal) from any voxel in the cluster to any other voxel in the cluster. Clusters are often taken to represent a set of neurons all involved in some single computation. They can also serve as the basis for *functional ROI*.

17.3.4 coregistration

The process of bringing two brain images into alignment ideally, you'd like them lined up so that their edges line up and the point represented by a given voxel in one image represents the same point in the other image. Coregistration generally refers specifically to the problem of aligning two images of different modalities - say, T1 fMRI images and PET images, or anatomical MRI scans and functional MRI scans. It goes for some of the same goals as *realignment*, but it generally uses different algorithms to make it more robust. See Coregistration FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#coregistration) for more.

17.3.5 contrast image

A voxel-by-voxel summary of the value of some *contrast* you've defined. This is often created as a voxel-by-voxel weighted sum of *beta images*, with the weights given by the value of the contrast vector. In SPM, it's actually written out as a separate image file; in other programs, it's usually written as a separate sub-bucket or the equivalent. It shouldn't be confused with the statistic image, which is a voxel-by-voxel of the test statistic associated with each contrast value. (In SPM, those statistic images are labeled spmT or spmF images.) **Only the contrast images - not the statistic images - are suitable for input to a second-level group analysis**. See Contrasts FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#contrasts) for more info on contrasts, and Random and Fixed Effects FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#random-and-fixed-effects) for more info on group analyses.

17.3.6 conjunction analysis

A way of combining contrasts, to look for activations that are shared between two conditions as opposed to differing between two conditions. It's implemented in SPM and other packages as essentially a logical AND-ing of contrasts - a way of looking for all the areas that are active in *both* one contrast and another. It's tricky to implement at the group level, though. Look at Contrasts FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#contrasts) for more info, and possibly Random and Fixed Effects FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#random-and-fixed-effects) as well.

17.3.7 contrast

The actual signal in fMRI data is unfortunately kind of arbitrary. The numbers at each voxel in your functional images don't have a whole lot of connection to any physiological parameter, and so it's hard to look at a single functional image (or set of images) and know the state of the brain. On the other hand, you can easily look at two functional images and see what's different between them. If those functional images are taken during different experimental conditions, and the difference between them is big enough, then you know something about what's happening in the brain during those conditions, or at least you can probably write a paper claiming you do. Which is good! So the fundamental test in fMRI experiments is not done on individual signal values or *beta weights*, but rather on differences of those things. A contrast is a way of specifying which images you want to include in that difference. A given contrast is specified as a vector of weights, one for each experimental condition / column in your design matrix. The contrast values are then created by taking a weighted sum of *beta weights* at each voxel, where the weights are specified by the contrast vector. Those contrast values are then tested for statistical significance in a variety of ways. Check out Contrasts FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#contrasts) for more info on contrasts in fMRI.

17.3.8 cutoff period

The longest length of time you want to preserve with your *high-pass filter*. A high-pass filter attentuates low frequencies, or slow oscillations; everything that repeats with a period slower than two minutes, say, you might reject as being clearly unrelated to your experiment. The cutoff period would be two minutes in the example above; it's the longest length of time you could possibly be interested in for your experiment. You generally want to set it to be way longer than an individual trial or block, but short enough to knock out most of the low-frequency noise. See Temporal Filtering FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#temporal-filtering) for more.

17.3.9 cytoarchitectonic

Relating to the look/type/architecture of individual cells. Not all neurons look exactly the same, and they're not all organized in exactly the same way throughout the brain. You can look in the brain and find distinct places where the "type" of neuron changes from one to another. You might theorize that a cell-level architecture difference might relate to something difference in the functions subserved by those cells. That's exactly what Brodmann theorized, and his *Brodmann areas* are based on cytoarchitectonic boundaries he found in the brain. Check out ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) for how cytoarchitectonic differences can be used

17.4 D

17.4.1 deconvolution

A mathematical operation in which the values from one function are removed from the values of another. In fMRI, where the signal is generally interpreted to be the result of a neuronal timeseries (which is modeled by the design matrix) convolved with a hemodynamic response function (which is modeled by a *canonical HRF*, *basis function*, or a *FIR (Finite Impulse Response) model*), the operation is usually used to separate the contributions of those two functions. SPM's *psychophysiological interaction (PPI)* function attempts to model the interaction of neuronal timeseries (as opposed to fMRI timeseries) by first deconvolving the canonical HRF and then checking the interaction at the neuronal, rather than hemodynamic level.

17.4.2 design matrix

A model of your experiment and what you expect the neuronal response to it to be. In general represented as a matrix (funnily enough), where each row represents a time point / TR / functional image and each column represents a different experimental effect. It becomes the model in a multiple regression, following the vector equation: Y = BX + E. Y is a vector of length a (equal to nframes from the scanner), usually representing the signal from a single voxel. B is a vector of b, representing the effect sizes for each of b experimental conditions. E is an error vector the same length as Y. X is your design matrix, of size a x b. Check out Basic Statistical Modeling FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#basic-statistical-modeling) for more.

17.4.3 detrending

There are multiple sources of noise in fMRI - head movement, transient scanner noise, gradual warming of the RF coils, etc. Many of them are simple, gradual changes in signal over the course of the session - a *drift* that can be linear, quadratic, or some higher polynomial that has very low frequency. Assuming that you don't have any experimental effect that varies linearly over the whole experiment, then, simply removing any very low-frequency drifts can be a very effective way of knocking out some noise. Detrending is exactly that - the removal of a gradual trend in your data. It often refers simply to linear detrending, where any linear effect over your whole experiment is removed, but you can also do a quadratic detrending, cubic detrending, or something else. Studies have shown that you're not doing much good after a quadratic detrending - most of the gradual noise is modeled well by a linear and/or quadratic function.

17.4.4 Diffusion Tensor Imaging (DTI)

A relatively newer technique in MRI that highlights white matter tracts rather than gray matter. It can be used to derive maps showing the prevailing direction of white matter fibers in a given voxel, which has given rise to a good deal of interest in using to derive connectivity data. Check out Connectivity FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#connectivity) for more.

17.4.5 dispersion derivative

The derivative with respect to the dispersion parameter in a gamma function. In SPM, the dispersion derivative of the *canonical HRF* looks a lot like the HRF but can be used as a *basis function*, to model some uncertainty in how wide you expect the HRF to be at each voxel.

17.4.6 drift

Some noise in an fMRI signal that is extremely gradual, usually varying linearly or quadratically over the course of a whole run of the scanner. This noise is usually called a drift, or a scanner drift. Sources of drifts are generally from the scanner - things like gradual warming of the magnet, gradual expansion of some physical element, etc. - but can also come from the subject, as in a gradual movement of the head downwards. Drifts often comprises a substantial fraction of the noise in a session, and can often be substantially removed by *detrending*.

17.4.7 dropout

The fMRI signal is contingent on having an extremely even, smooth, homogenous background magnetic field and a precisely calculated gradient field. If anything distorts the background field or the gradient field in a localized fashion, the signal in that region can drop to almost nothing due to the distortions. This is called dropout or signal dropout. This is most common in regions of high *susceptibility* - brain regions near air/tissue interfaces, where the differing magnetic signatures of the two materials causes major local distortions. In those regions, it's difficult to get much signal from the scanner, and *Signal-to-Noise Ratio (SNR)* shrinks drastically, meaning it's hard to find activations there. A good deal of research has been done to ameliorate dropout; recently, it's been shown spiral in-out imaging does a pretty good job avoiding dropout in the traditionally bad regions. See Scanning FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#scanning) for more.

17.4.8 Dynamic Causal Modeling (DCM)

A new statistical analysis technique for making inferences about *functional connectivity*. It allows the user to specify a small set of *functional ROI* and a design matrix, and then given some data, produces a set of connectivity parameters. These parameters include both a "default" measure of connectivity between the ROIs, as well as a dynamic measure of how that connectivity changed across the experiment - specifically, whether any experimental effect changed the connectivity between regions. Has been used, for example, to investigate whether category effects in vision are modulated by bottom-up or top-down pathways. See Connectivity FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#connectivity) for much more.

17.5 E

17.5.1 Echo-planar Imaging (EPI)

A type of pulse sequence in which lines of *k-space* are sampled in order. This is the more conventionally-used pulse sequence around the world, and has some advantages over other sequences of being slightly easier to analyze and pretty fast. It is quite susceptible to various *artifact* and distortions, though. Check out Scanning FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#scanning) for more.

17.5.2 EEG (Electroencephalogram)

Stands for electroencephalogram. A neuroimaging technique in which electrodes are pasted to the skull to directly record the electrical oscillations caused by neuronal activity - sometimes called "brain waves". Allows the recording of electrical activity at millisecond resolution, far better than PET or fMRI, but suffers from a lack of regional specificity, as it's extremely difficult to tell where in the brain a given EEG signal originated. The exact nature of the neuronal activity that gives rise to the EEG signal is not entirely clear, but active efforts are underway at several facilities to combine EEG and fMRI to try and get excellent spatial and temporal resolution in the same experiment. See also *Event-related Potential (ERP)* below.

17.5.3 effective connectivity

A term introduced by Karl Friston (https://en.wikipedia.org/wiki/Karl_J._Friston) in order to highlight the difference between "correlational" methods of inferring brain connectivity and the actual concept of causal connection between brain areas. The distinction made is one between correlation and causation. Effective connectivity (EC) stands in contrast to *functional connectivity*, which goes more with correlation. EC between brain areas is defined as "the influence one neural system exerts over another either directly or indirectly." It doesn't imply a direct physical connection - simply a causative influence. It's a lot harder to establish that two regions are effectively connected than it is to establish that they're functionally connected, but EC supports more interesting inferences than FC does.

17.5.4 efficiency

A statistical concept in experimental design, used to describe how accurately one can model the shape of a response. It's at the other end of a tradeoff with *power*, which is used to describe how well you can detect any effect at all. Block experiments are very low in efficiency; because the trials come on top of each other, it's difficult to tell how much signal comes from one trial and how much from another, so the shape is muddled. Fully-randomized event-related experiments have high efficiency; you can sample many different points of the HRF and know exactly which HRF you're getting. Experiments that have very high power must necessarily have lower efficiency - you can't be perfect at both. Check Design FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#design) our for more on the efficiency/power tradeoff. Also check out Jitter FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#jitter) for how to maximize efficiency in your experiment.

17.5.5 Event-related Potential (ERP)

A variation on EEG in which you focus not on the ongoing progression of activity, but rather electrical activity in response to a particular stimulus (or lack thereof). Instead of looking at a whole EEG timecourse or frequency spectrum, you take a small window of time (1 second, say) after each presentation of a trial A, and average those windows together to get the average response to your stimulus A. This creates a *peristimulus timecourse*, not unlike that for an HRF in fMRI. You can then compare the time-locked average from one condition to that from another condition, or analyze a single time-locked average for its various early and late components. ERPs and the advent of a *event-related design* in fMRI allow the same designs to be used in both EEG and fMRI, presenting the promise of combining the two into one super-imaging modality which will grow out of control and destroy us all. Or not.

17.5.6 event-related design

An experimental design in which different trial types are intermixed throughout the experiment, usually in random or pseudo-random fashion. Contrasts with a *block design*, where trials of the same type are collected into chunks. Event-related designs sacrifice *power* in exchange for higher *efficiency*, as well as psychological unpredictability, which allow new kinds of paradigms in fMRI. Check out Design FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#design) for way more about event-related designs, and Jitter FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#jitter) for why randomization is all the rage amongst the kiddies.

17.6 F

17.6.1 F-contrast

A type of *contrast* testing a F-statistic, as opposed to a t-statistic or something else. Allows you to test several linear constraints on your model at once, joining them in a logical OR. In other words, it would allow you to test the hypothesis that A and B are different OR A and C are different OR B and C are different at a given voxel. Another way of describing that would be to say you're testing whether there are any differences among A, B and C at all. F-contrasts can be tricky (if not impossible) to bring forward to a random-effects group analysis. See Contrasts FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#contrasts) and Random and Fixed Effects FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#random-and-fixed-effects) for more.

17.6.2 False Discovery Rate (FDR)

A statistical concept expressing the fraction of accepted hypotheses in some large dataset that are false positives. The idea in controlling FDR instead of *Family-wise error correction (FWE)* is that you accept the near-certainty of a small number of false positives in your data in exchange for a more liberal, flexible, reasoned correction for multiple comparisons. Since most researchers accept the likelihood of a small amount of false positives in fMRI data anyways, FDR control seems like an idea whose time may have arrived in neuroimaging. Check out P threshold FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#p-threshold) for more.

17.6.3 Family-wise error correction (FWE)

In a dataset of tens of thousands of voxels, how do you decide on a statistical threshold for true activation? The scientific standard of setting the statistic such that $p < 0.05$ isn't appropriate on the voxel level, since with tens of thousands of voxels you'd be virtually guaranteed hundreds of false positives - voxels whose test statistic was highly improbably just by chance. So you'd like to correct for multiple comparisons, and you'd like to do it over the whole data set at once - correcting the family-wise error. Family-wise error correction methods allow you to set a global threshold for false positives; if your family-wise threshold is $p < 0.05$, you're saying there's a 95% chance there are

NO false positives in your dataset. There are several accepted methods to control family-wise error: Bonferroni, various Bonferroni-derived methods, *Gaussian random field*, etc. FWE stands in contrast to *False Discovery Rate (FDR)* thresholding, which threshold the *number* of false positives in the data, rather than the chance of *any* false positives in the data. See P threshold FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#p-threshold) for more.

17.6.4 FIR (Finite Impulse Response) model

A type of design matrix which assumes nothing about the shape of the *Hemodynamic Response Function (HRF)*. With an FIR model, you don't convolve your design matrix with a *canonical HRF* or any *basis function*. Instead, you figure out how long an HRF you'd like to estimate - maybe 10 or 15 TRs following your stimulus. You then have a separate column in your design matrix for every time point of the HRF for every different condition. You separately estimate *beta weights* for every time point, and then line them up to form the timecourse of your HRF. The advantage is that you can separately estimate an unbiased HRF at every voxel for every condition - tremendous flexibility. The disadvantage is that the confidence in any one of your estimates will drop, because you use so many more degrees of freedom in estimation. Full FIR models may not be useable for very complex experiments or certain types of designs. Check out Percent Signal Change FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#percent-signal-change) for more on FIR models.

17.6.5 fishing expedition

What happens when your data doesn't really offer any compelling or interpretable story about your task... so you try every conceivable way of analyzing it and every conceivable contrast possible to find something interesting looking. Then, of course, it behooves you to write your paper as if you'd been looking for that all along.

17.6.6 fixed-effects

An analysis that assumes that the subjects (or scanning sessions, or scanner runs, or whatever) you're drawing measurements from are fixed, and that the differences between them are therefore not of interest. This allows you to lump them all into the same design matrix, and consider only the variance between timepoints as important. This allows you to gain in power, due to the increased number of timepoints you have (which leads to better estimates and more degrees of freedom). The cost is a loss of inferential power - you can only make inferences in this case about the actual group of subjects (or scanner sessions, or whatever) that you measured, as opposed to making inferences about the population from which they were drawn. Making population inferences requires analyzing the variance between subjects (/scanner/sessions... you get the idea) and treating them as if they were drawn randomly from a population - in other words, a random-effects analysis. Check out Random and Fixed Effects FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#random-and-fixed-effects) for more.

17.6.7 fixed ISI

Stands for fixed inter-stimulus interval. A type of experiment in which the same time separates the beginning of all stimuli - trials needn't be all exactly the same length, but the onsets of stimuli are all separated by exactly the same amount of time. *Event-related design* or *block design* experiments can be fixed ISI. fixed ISI event-related experiments, though, are pretty bad at both *efficiency* and *power*, especially as the ISI increases. In general, several empirical studies have shown that for event-related designs, *variable ISI* is the way to go. For block designs, the difference is fairly insignificant, and variable ISI can make the design less powerful, depending on how it's used. See Jitter FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#jitter) for more on the difference between fixed and variable.

17.6.8 flattening

One inconvenient thing about mapping the brain is the way that it's all folded and scrunched into that little head like so much wadded-up tissue. Voxels that appear to be neighboring, for example, might in fact be widely separated on the cortical sheet, but have that distance obscured by the folds of a gyrus in between them. In order to study the spatial organization of a particular cortical region, it may then be useful to "unfold" the brain and look at it as if the cortical sheet had been flattened out on a table. Indeed, some phenomena like retinotopy are near-impossible to find without cortical flattening. Several software packages, then, allow you to create a surface map of the brain - a 3D graphical representation fo the cortical surface - and then apply several automated algorithms to flatten it out, and project your functional activations onto the flattened representation. FreeSurfer is best known for this type of analysis.

17.6.9 fMRI

Stands for functional magnetic resonance imaging. The small 'f' is used to distinguish functional MRI, often used for scanning brains, from regular old static MRI, used for taking pictures of knees and things. Check out Physiology and fMRI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#physiology-and-fmri) for more info on the physics and theory behind fMRI, or Scanning FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#scanning) for useful (with any luck) answers about how to set parameters for your experiment.

17.6.10 FreeSurfer

FreeSurfer (http://freesurfer.net/) is a brain imaging software package developed by the Athinoula A. Martinos Center for Biomedical Imaging at Massachusetts General Hospital for analyzing MRI data. It is an important tool in functional brain mapping and facilitates the visualization of the functional regions of the highly folded cerebral cortex. It contains tools to conduct both volume based and surface based analysis, which primarily use the white matter surface. FreeSurfer includes tools for the reconstruction of topologically correct and geometrically accurate models of both the gray/white and pial surfaces, for measuring cortical thickness, surface area and folding, and for computing inter-subject registration based on the pattern of cortical folds. In addition, an automated labeling of 35 non-cortical regions is included in the package. (Taken from Wikipedia: FreeSurfer (https://en.wikipedia.org/wiki/FreeSurfer))

17.6.11 FSL (FMRIB Software Libraryand)

FSL (http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/FSL) is a comprehensive library of analysis tools for fMRI, MRI and DTI brain imaging data. It runs on Apple and PCs (both Linux, and Windows via a Virtual Machine), and is very easy to install. Most of the tools can be run both from the command line and as GUIs. For an overview of the algorithms included in FSL go to the overview section (http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/FslOverview) on their homepage.

17.6.12 Fourier basis set

A particular and special type of *basis function*. Instead of using a standard *design matrix*, an analysis with a Fourier basis set simply uses a set of sines or cosines of varying frequency for the design matrix columns for each condition. Because a combination of cosines can be used to model almost any periodic function at all, this design matrix is extremely unbiased - in particular as to when your activations took place, since you don't have to specify any onsets. You simply let your software estimate the best match to the period parts of your signal (even if they're infrequent). This allows you, like an *FIR (Finite Impulse Response) model*, to estimate a separate HRF for every voxel and every condition, as well as come up with detailed maps of onset lag at each voxel and other fun stuff. The disadvantages of this model include relatively lower power, due to how many degrees of freedom are used in the basis set, and some limitations on what functions can be modeled (edge effects, etc.) It also requires you to use an *F-contrast* to test it, since the individual parameters have no physiological interpretation.

17.6.13 functional connectivity

A term introduced by Karl Friston (https://en.wikipedia.org/wiki/Karl_J._Friston) to highlight the differences between "correlational" methods of inferring brain connectivity and the causational concepts and inferences that you might want to make. The difference is between correlation and causation; functional connectivity is more correlational. Brain regions which are functionally connected merely must have some sort of correlation in their signal, rather than having any direct causal influence over each other. This is in contrast to *effective connectivity*, which demands some causation be included. Functional connectivity is rather easier to establish, but supports perhaps less interesting inferences. Most methods out there looking at connectivity are good only for functional connectivity, with TMS being a notable exception. See Connectivity FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#connectivity) for more.

17.6.14 functional ROI

Any region-of-interest (ROI) that is generated by looking at functional brain activation data is considered a functional ROI. It may also have reference to anatomical information; you may be looking for all active voxels within the amygdala, say. That would be both an anatomical and functional ROI. Any subsset of voxels generated from a list of functionally active voxels, though, can comprise a functional ROI. See ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) for ways you can use 'em.

17.7 G

17.7.1 Gaussian random field

Whoo, that's a heck of a way to start a letter. Essentially, a type of random field (https://en.wikipedia.org/wiki/Random_fields) that satisfies a Gaussian distribution, I guess. As it applies to fMRI, the key thing to know is that SPM's default version of *Family-wise error correction (FWE)* operates by assuming your test statistics make up a Gaussian random field and are therefore subject to several inferences about their spatial distribution. FWE correction based on Gaussian random fields has been shown to be conservative for fMRI data that has not been smoothed rather heavily. See P threshold FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#p-threshold) for more info.

17.7.2 general linear model (GLM)

The general linear model is a statistical tool for quantifying the relationship between several independent and several dependent variables. It's a sort of extension of multiple regression, which is itself an extension of simple linear regression. The model assumes that the effects of different independent variables on a dependent variable can be modeled as linear, which sum in a standard linear-type fashion. THe standard GLM equation is Y = BX + E, where Y is signal, X is your *design matrix*, B is a vector of *beta weights*, and E is error unaccounted for by the model. Most neuroimaging software packages use the GLM as their basic model for fMRI data, and it has been a very effective tool at testing many effects. Other forms of discovering experimental effects exist, notably non-model-based methods like *principal components analysis (PCA)*. Check out Basic Statistical Modeling FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#basic-statistical-modeling) for more info on how the GLM is used in fMRI analysis.

17.7.3 GitHub

GitHub (https://github.com/) is a Git repository web-based hosting service that offers distributed revision control and source code management (SCM). GitHub is a web-based graphical interface that allows programmers to develope and contribute code together. For more, see Wikipedia's GitHub page (https://en.wikipedia.org/wiki/GitHub) or go to the offical homepage (https://github.com/).

17.7.4 global effects

Any change in your fMRI signal that affects the whole brain (or whole volume) at once. Sources of these effects can be external (scanner *drift*, etc.) or physiological (motion, respiration, etc.). They are generally taken to be non-neuronal in nature, and so generally you'd like to remove any global effects from your signal, since it's extremely unlike to be caused by any actual neuronal firing. See Physiology and fMRI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#physiology-and-fmri) and Realignment FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#realignment) for thoughts on how to account for global effects in your dataset.

17.7.5 global scaling

An analysis step in which the voxel values in *every image* are divided by the global mean intensity of *that image*. This effectively makes the global mean identical for every image in the analysis. In other words, it effectively removes any differences in mean global intensity between images. This is different than *grand mean scaling*! Global scaling (also called proportional scaling) was introduced in PET, where the signal could vary significantly image-to-image based on the total amount of cerebral blood flow, but it doesn't make very much sense to do generally in fMRI. The reason is because if your activations are large, the timecourse of your global means may correlate with your task - if you have a lot of voxels in the brain going up and down with your task, your global mean may well be going up and down with your task as well. So if you divide that variation out by scaling, you will lose those activations and possibly introduce weird negative activations! There are better ways to take care of *global effects* in fMRI (see Physiology and fMRI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#physiology-and-fmri) for some), considering that moment-to-moment global variations are very small in fMRI compared to PET. They can be quite large session-to-session, though, so *grand mean scaling* is generally a good idea.

17.7.6 grand mean scaling

An analysis step in which the voxel values in every image are divided by the average global mean intensity of the *whole session*. This effectively removes any mean global differences in intensity between sessions. This is different than *global scaling*! This step makes a good deal of sense in fMRI, because differences between sessions can be substantial. By performing it at the first (within-subject) level, as well, it means you don't have to do it at the second (between-subject) level, since the between-subject differences are already removed as well. This step is performed by default by all the major analysis software packages.

17.7.7 Granger causality, Granger causality modeling

A statistical concept imported from econometrics intended to provide some new leverage on tests of *functional connectivity*. Granger causality is somewhat different from regular causality; testing Granger causality essentially boils down to testing whether information about the values or lagged values of one timecourse give you any ability to predict the values of another timecourse. If they do, then there's some degree of Granger causality. The concept is still somewhat controversial in econometrics, and the same goes for neuroimaging. What's clear is the test is still effectively a correlational test, though far more sophisticated than just a standard cross-correlation. So establishing Granger causality between regions is enough to establish *functional connectivity* and some degree of temporal precedence, but probably not enough to establish *effective connectivity* between those regions. Check out Connectivity FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#connectivity) for more.

17.8 H

17.8.1 hand-waving

An explanatory technique frequently used in fMRI research to obscure the fact that no one really knows what the hell is going on.

17.8.2 Hemodynamic Response Function (HRF)

When a set of neurons in the brain becomes more active, the brain responds by flooding the area with more highly-oxygenated blood, enabling an MRI scanner to detect the *BOLD (blood oxygen level-dependent) signal* contrast in that region. But that "flooding" process doesn't happen instantaneously. In fact, it takes a few seconds following the onset of neuronal firing for BOLD signal to gradually ramp up to a peak, and then several more seconds for BOLD signal to diminish back to baseline, possibly undershooting the baseline briefly. This gradual rise followed by gradual fall in BOLD signal is described as the hemodynamic response function. Understanding its shape correctly is crucial to analyzing fMRI data, because the neuronal signals you're looking to interpret aren't directly present in the data; they're all filtered through this temporally extended HRF. A great deal of statistical thought and research has gone into understanding the shape of the HRF, how it sums over time and space, and what physiological processes give rise to it. Check out HRF FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#hrf) for more about how it's modeled in fMRI analysis.

17.8.3 hierarchical model

A type of *mixed-effects* model in which both random and fixed effects are modeled but separated into different "compartments" of "levels" of the modeling. The standard group model approach in fMRI is hierarchical - you model all the fixed (within-subjects) effects first, then enter some summary of those fixed effects (the *beta weights* or *contrast image*) into a *random-effects* model, where all the random (between-subject) effects are modeled. This allows separate treatment of the between- and within-subject variance. Check out Random and Fixed Effects FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#random-and-fixed-effects) for more info.

17.8.4 high-pass filter

A type of frequency filter which "passes through" high frequencies and knocks out low frequencies. Has the effect, therefore, of reducing all very low frequencies in your data. Since fMRI noise is heavily weighted towards low frequencies, far lower than the frequencies of common experimental manipulations, high-pass filters can be a very effective way of removing a lot of fMRI noise at little cost to the actual signal. Setting the *cutoff period* is of crucial importance in high-pass filter construction. Contrasts with *low-pass filter* and *band-pass filter*. See Temporal Filtering FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#temporal-filtering) for more info.

17.9 I

17.9.1 Impulse Response Function (IRF)

In linear systems theory, you can predict a system's response to any arbitrary stimulus if you a) assume that its response to stimuli obeys certain assumptions about linearity (summation, etc.) and b) you know how the system responds to a single instantaneous impulse stimulus. The system's response in this case is called the IRF, or impulse response function. Many analyses - the *general linear model (GLM)*, primarily - of the brain's response to stimuli proceed along linear systems methods, assuming that the IRF is equivalent to the hemodynamic response function (HRF). This HRF can be measured or simply assumed. IRF and HRF are sometimes used interchangeably in fMRI literature.

17.9.2 Independent Components Analysis (ICA)

A statistical technique for analyzing signals that are presumed to have several independent sources mixed into the single measure signal. In fMRI, it's used as a way of analyzing data that doesn't require a model or *design matrix*, but rather breaks the data down into a set of statistically independent components. These components can be then (hopefully) be localized in space in some intelligible way. This enables you, theoretically, to *discover* what effects were "really" present in your experiment, rather than hypothesizing the existence of some effects and testing the significance of your hypothesis. It's been used more heavily in *EEG (Electroencephalogram)* research, but is beginning to be applied in fMRI, although not everything about the results it gives is well understood. Its use in *artifact* detection is clear, though. It differs from *principal components analysis (PCA)*, an algorithm with similar goals, because the components it chooses have maximal statistical independence, rather than maximizing the explained variance of the dataset.

17.9.3 inflation

Related to *flattening*. A downer about superimposing activation results on the brain is that brains are kind of inconveniently wrinkled up. This makes it difficult to see the exact spatial relationship of nearby activations. Two neighboring voxels might well be separated by a large distance on the cortical sheet, but one is buried deep in a sulcus and one is on top of a gyrus. Inflation and flattening are visualization techniques that aim to work around that problem. Inflation works by first doing *surface mapping* to construct a 3-D model of the subject's cortical surface, and then applies graphics techniques to slowly blow up the brain, as if inflating it. This gradually reduces the wrinkling, spreading out the sulci and gyri until, ultimately, you could inflate the brain all the way to spherical shape. Usually inflation stops when most of the smaller sulci and gyri are flattened out, as this allows much nicer visualization of phenomena like retinotopy.

17.9.4 Interfaces

Interfaces in the context of Nipype are program wrappers that allow Nipype which runs in Python (https://www.python.org/) to run a program or function in any other programming language. As a result, Python (https://www.python.org/) becomes the common denominator of all neuroimaging software packages and allows Nipype to easily connect them to each other. For a full list of software interfaces supported by Nipype go here (http://nipype.readthedocs.io/en/latest/documentation.html). For more see the introduction section of this beginner's guide (http://miykael.github.io/nipype-beginner-s-guide/nipype.html#interfaces).

17.9.5 Inter-stimulus Interval (ISI)

The length of time in between trials in an experiment. Usually measure from the onset of one trial to the onset of the next. The length and variability of your ISI are crucial factors in determing how much *power* and *efficiency* your experimental design provide, and thus how nice your results will look. See Design FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#design) and Jitter FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#jitter) for info about figuring out the proper length of your ISI.

17.9.6 IPython

IPython (http://ipython.org/) is an interactive interpreter for the Python (https://www.python.org/) language. At the beginning it was only a command shell but with time and with the introduction of Jupyter Notebook (http://jupyter.readthedocs.org/en/latest/) becomes more and more the best Python (https://www.python.org/) computational environment at hand. IPython is capable to compute in multiple programming languages and offers enhanced introspection, rich media, additional shell syntax, tab completion, and rich history. For more, go to IPython's offical homepage (http://ipython.org/).

17.9.7 isotropic

The same size in all directions. A sphere is isotropic. An ovoid is not. Isotropy is the degree to which something is isotropic. Smoothing kernels are often isotropic, but they don't have to be - they can be *anisotropic*. *Voxels* are often anisotropic originally, but are resample to be isotropic later in processing.

17.10 J

17.10.1 jittered

A term used to describe varying the *Inter-stimulus Interval (ISI)* during your experiment, in order to increase *efficiency* in the experimental design. Can also be used (although less frequently these days) to describe offsetting the TR by a small amount to avoid trial lengths being an exact multiple of the TR. Used as a noun - "I made sure there was some jitter in my design" - or a verb - "We're going to jitter this design a little." Check out Jitter FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#jitter) for all the gory details.

17.11 K

17.11.1 k-space

One way to take a 3-D picture would be to sample various points in space for the intensity of light there, and then reassemble those samples into a volume - an easy reassembly process, since the sampled intensity is exactly what you want to see. But that's not how MRI scanners take their pictures. Instead of sampling real space for the intensity of light at a given point, they sample what's called k-space. A given point in k-space describes both a frequency and a direction of oscillation. Very low frequencies correspond to slow oscillations and gradual changes in the picture at that direction; higher frequncies correspond to fast oscillations and sharp changes (i.e., edges) in the picture at that direction. The points in k-space don't correspond to any real-world location! They correspond only to frequency and direction. This is the space that MRI scanner samples. K-space can be sample in different patterns; these correspond to different *pulse sequence* at the scanner.

17.11.2 kernel

See *smoothing kernel*.

17.12 L

17.12.1 linear drift

See *drift*.

17.12.2 localizer

One way of dealing with the sizeable differences in brain anatomy between subjects is to use an analysis that focuses on regions of interest, rather than individual voxels. The danger in using anatomically defined regions of interest is that the mapping between function and anatomy varies widely between subjects, so one subject might activate the whole calcarine sulcus during a visual stimulus and another might only activate a third of it. One way around this variability is to use functionally-defined regions of interest. A localizer task is one designed to find these functional ROI. The idea is to design a simple task that reliably activates a particular region in all or most subjects, and use the set of voxels activated by that localizer task as an ROI for analyzing another task. The simple task is called a localizer because it is designed to localize activation to a particular set of voxels within or around an anatomical structure. See ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) for more on the region-of-interest approach.

17.12.3 long event-related designs

An experimental design in which single trials are the basic unit, and those single trials are separated by enough time to allow the *Hemodynamic Response Function (HRF)* to fully return to baseline before the next trial - usually 20-30 seconds. This design is a subtype of a *event-related design*, contrasting with the other subtype, *rapid event-related designs*. Long event-related designs have the advantage of being very straightforward to analyze, and incredibly easy to extract timecourses from. They have the disadvantage, though, of having many fewer trials per unit time than a *block design* or rapid event-related design, and so long event-related designs are both very low-powered and very inefficient. They're not widely used in fMRI any more, unless the experiment calls for testing assumptions about *Hemodynamic Response Function (HRF)* summation or something. See Design FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#design) for more.

17.12.4 low-pass filter

A type of filter that "passes through" low frequencies and suppresses high frequencies. This has the effect of smoothing your data in the temporal (rather than spatial) domain - very fast little jiggles and quick jumps in the signal are suppressed and the timecourse waveform is smoothed out. If temporal-domain noise is random and independent across time, low-pass filtering helps increase *Signal-to-Noise Ratio (SNR)* ratio in the same way *spatial smoothing* does. But, unfortunately, fMRI temporal-domain noise is highly colored, and so low-pass filtering usually ends up suppressing signal. Check out Temporal Filtering FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#temporal-filtering) for lots more on the low-pass filtering controversy.

17.13 M

17.13.1 MapNode

See *Workflow*.

17.13.2 mask, mask image

A special type of image file used in *SPM (Statistical Parametric Mappin)* (and other programs) which is used to specify a particular region of the brain. Every voxel in that region has intensity 1; everything outside of that region has intensity 0. Such an image is also called a binarized map. You might have a *Region of Interest (ROI)* mask, to specify the location of a ROI, or you might have a brain mask, where the mask shows you where all of the in-brain voxels are (so that you can analyze only the in-brain voxels, for example). Most ROI programs that create image files create masks. SPM standardly creates a mask image file based on intensity thresholds during model estimation, and only estimates voxels within its brain mask.

17.13.3 mat file (or dot-mat file, .mat file, etc.)

1. A *MATLAB* file format which contains saved Matlab variables, and allows you to save variables to disk and load them into the workspace again from disk. Format is binary data, so it's not accessible with text editors.

2. One special kind of .mat file in SPM is the .mat file which can go along with a format .img/.hdr pair. A .mat file with the same filename as a .hdr/.img pair is interpreted in a special way by SPM; when that image file is read, SPM looks into the .mat file for a matrix specifying a position and orientation transform of the image. In this way, SPM can save a rigid-body transformation of the image (rotation, zoom, etc.) without actually changing the data in the .img file. Almost every SPM image-reading function automatically reads the .mat file if it's present, and many functions which move the image around (*realignment*, *slice timing*, etc.) give you the option to save the changes as a .mat file instead of actually re-slicing the image.

17.13.4 MATLAB

The dominant software package in scientific and mathematical computing and visualization. Originally built to do very fast computations and manipulations of very large arbitrary matrices; now includes things like a scripting language, graphical user interface builder, extensive mathematical reference library, etc. See MATLAB Basics FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#matlab-basics) for basic information on how to use MATLAB. For everything else, check out the Matlab Documentation (http://www.mathworks.com/help/).

17.13.5 mental chronometry

See *chronometry* or Mental Chronometry FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#mental-chronometry).

17.13.6 microanatomy

A level of anatomical detail somewhere around and above *cytoarchitectonic*, but smaller than the standard anatomic strucures. This level of detail refers to things like cell type, or the organization of cell layers and groups. See ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) for information on using microanatomical detail in your study.

17.13.7 mixed-effects

A model which combines both *fixed-effects* and *random-effects*. Most fMRI group effects model are mixed-effects models of a special type; they are generally hierarchical, where the fixed effects and random effects are partitioned and evaluated separately. Check out Random and Fixed Effects FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#random-and-fixed-effects) for more info.

17.13.8 MNI space, MNI templates

The Montreal Neurological Institute (MNI) has published several "template brains," which are generic brain shapes created by averaging together hundreds of individual anatomical scans. The templates are blurry, due to the averaging, but represent the approximate shape of an "average" human brain. One of these templates, the MNI152, is used as the standard *normalization* template in SPM. This differs from *Talairach* normalization, which uses the *Talairach* brain as a template. So normalized SPM results aren't quite in line with Talairach-normalized results. The MNI brain differs slightly from the Talairach brain in several ways, particularly in the inferior parts of the brain. In order to report normalized SPM results in Talairach coordinates for ease of reference, it's necessary to convert the MNI coordinates into Talairach space with a script called mni2tal.m from Matthew Brett. See ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) and Normalization FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#normalization) for more.

17.13.9 motion correction

See *realignment*.

17.13.10 mutual information

A concept imported from information theory into image analysis. If you have two random variables, A and B, and would like to quantify the amount of statistical dependence between them, one way you might do it is by asking: how much *more* certain are you about the value of B if you know the value of A? That amount is the amount of mutual information between A and B. In more precise terms, it's the distance (measured by a K-L statistic) between the joint probability distribution P(ab) and the product of their individual distributions, P(a) * P(b). It comes up in fMRI primarily in *coregistration*. Mutual information-based methods provide a much more robust way of lining up two images than simple intensity-based methods do, and so most current coregistration programs use it or a measure derived from it. See Coregistration FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#coregistration) for more info.

17.14 N

17.14.1 NIfTI

NIfTI (http://nifti.nimh.nih.gov/) stands for Neuroimaging Informatics Technology Initiative and is a file format most commenly used in neuroimaging. For more information see this blog (http://brainder.org/2012/09/23/the-nifti-file-format/).

17.14.2 Nipype

Nipype stands for Neuroimaging in Python - Pipelines and Interfaces and is this amazing software package for which this beginner's guide is written for. For more information go to the introductory page (http://miykael.github.io/nipype-beginner-s-guide/nipype.html) of this guide.

17.14.3 neurological convention

Radiological images (like fMRI) that are displayed where the left side of the image corresponds to the left side of the brain (and vice versa) are said to be in "neurological convention" or "neurological format." In radiological convention, left is right and right is left. Those crazy radiologists.

17.14.4 Node

See *Workflow*.

17.14.5 normalization

A spatial preprocessing technique in which anatomical and/or functional MRI images are warped in order to more closely match a template brain. This is done in order to reduce intersubject variability in brain size and shape. The warping can be affine in nature or nonlinear, and can be done on a voxelwise basis or with respect to the surfaces of the brains only. All the major neuroimaging packages support some form of normalization, but there are many questions about how much variability it actually removes. See Normalization FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#normalization) for more answers than you can shake a stick at, and even more questions than that.

17.15 O

17.15.1 onset

In order to create a *design matrix* for your experiment, you need to know when, in time, each of your trials started and how long they lasted. The beginning of a trial is commonly called an onset. An onset vector is a list of starting times for the trials of a particular condition. If you have 15 trials in condition A, your onset vector for condition A will have 15 numbers, each one specifying the moment in time when a particular trial started. The times are usually specified in either seconds or in TR. Generally all neuroimaging software packages require you to enter your onset vectors somehow, or construct a design matrix from them, as input before they can estimate a model. Check out Basic Statistical Modeling FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#basic-statistical-modeling) for more.

17.15.2 outlier

Any point in a dataset (of any kind) whose value lies wayyyyy outside the distribution of the rest of the points. Outliers are often removed from datasets in many scientific domains, because their extreme values can give them undue influence over the description of the data distribution; as one example, outliers can severely skew statistics like mean or variance. Figuring out just how far an outlier need be from the center of the distribution to be removed, though, is a tricky procedure, and often extremely arbitrary. Outlier detection and removal is one key aim of artifact detection schemes and programs.

17.15.3 orthogonal, orthogonalize, orthogonality

Orthogonal means perpendicular. Two things that are orthogonal to each other are perpendicular, to orthogonalize two things means to make them orthogonal, etc. The terms, though, are generally used less for real lines in space than for vectors. Any list of numbers can be taken to represent a point or a line in some space, and those lists of numbers can thus be made orthogonal by tweaking their elements such that the lines they represent become perpendicular. In more common terms, this corresponds to removing correlations between two lists of numbers. Two lists are "collinear" to the degree that they have some correlation in their elements, and they are orthogonal to the degree to that they have no correlation whatsoever in their elements. Two perfectly orthogonal lists have values that are totally independent of one another, and vice versa. Having columns in a *design matrix*, or elements in two contrasts, not be orthogonal can pose problems for estimating the proper *beta weights* for those columns or contrasts, so many programs either require certain structures be orthogonal or do their own orthogonalization when the issue comes up. Check out Basic Statistical Modeling FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#basic-statistical-modeling) and Contrasts FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#contrasts) for more info.

17.16 P

17.16.1 p-threshold

A particular probability value which is used as a threshold for deciding which voxels in a *contrast* are active and which are not. The contrast image is rendered in terms of some statistic, like a T or F, at each voxel, and each statistic can then be assigned a particular p-value - the likelihood that such a value would occur under the null hypothesis of no real activation. Voxels with p-values smaller than the threshold are declared active; other voxels are declared inactive. P-thresholds can be manipulated to account for multiple comparisons, spatial and temporal correlation, etc. See P threshold FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#p-threshold) for lots, lots more.

17.16.2 parameter weights

See *beta weights*.

17.16.3 partial voluming

In doing *segmentation*, a major problem in assigning a particular voxel to a tissue-type category or anatomical structure is that tissue and structure boundaries rarely line up exactly with voxel boundaries. So a given voxel might contain signal from two or more different tissue types. If one of the assumptions of segmentation is that different tissue types give off different signals (usually MR intensity), voxels with a mixture of tissue types pose a problem, because their intensity may lie in between the canonical intensity of any one tissue type. Oftentimes segmentation algorithms simply make a guess based on which tissue type the voxel seems closest to, but this can pose a problem in calculating, say, the total volume of gray matter in a brain. If half of your "white-matter" voxels have some gray matter in them, but you count them only as white matter, you're missing a whole lot of gray matter in your volume calculation. This is the partial volume problem, and a partial voluming effect is this type of tissue mixing. See Segmentation FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#segmentation) for more.

17.16.4 peak voxel

The most active voxel in a cluster, or the voxel in a cluster that has the highest test statistic (T-stat or F-stat or whatever). Often the coordinates of only the peak voxel are reported for a cluster in papers, and sometimes time-courses or *beta weights* are extracted only from the peak voxel. See ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) and Percent Signal Change FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#percent-signal-change) for more info on why that would be.

17.16.5 percent signal change

A measure of signal intensity that ignores the arbitrary baseline values often present in MR signal. A timecourse of signal can be viewed as a timecourse of changes from some baseline value, rendered in units of percent of that baseline value. The baseline is then chosen on a session-specific basis in some reasoned way, like "the mean of the timecourse over the whole session," or "the mean of the signal during all rest periods." This gets around the problem that MR signal is often scaled between sessions by some arbitrary value, due to how the scanner feels at that moment and the physiology of the subject. Two signal timecourses that are identical except for an arbitrary scaling factor will be totally identical when converted to percent signal change. Percent signal changes timecourses are thus used to show intensity timecourses from a given region or voxel during some experimental manipulation. Percent Signal Change FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#percent-signal-change) has everything you ever wanted to know about the measure, or at least everything I could think of before noon.

17.16.6 peristimulus timecourse

Means "with respect to the stimulus." A peristimulus timecourse is one that starts at the *onset* of a given stimulus. Sometimes a peristimulus timecourse will start with negative time and count down to a zero point before counting up again; the zero point is always the onset of a given stimulus. This is the same as a time-locked average timecourse. See Percent Signal Change FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#percent-signal-change) for more on why you would want to look at these.

17.16.7 perfusion

A type of fMRI imaging which doesn't look at BOLD contrast. Instead, blood is magnetically "labeled" just before it gets to the brain, and it's then tracked through the brain over time. Perfusion imaging has several advantages over BOLD - a different and flatter noise profile, possibly less variability over subjects, and a readily interpretable physiological meaning for the absolute units are chief among those. The major disadvantage is that *Signal-to-Noise Ratio (SNR)* is significantly smaller in perfusion imaging, at least in single subjects. This probably makes it less suitable for most current fMRI designs, but it may be a better option for novel designs (blocks lasting several minutes, for example). See Scanning FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#scanning) for a fuller discussion of the pros and cons of each.

17.16.8 permutation test

A type of statistical test, like a T-test or F-test, but one which assumes much less about the distribution of the random variable in question. This is a type of nonparametric test related to *bootstrapping*. It has significant advantages over standard parametric tests under certain conditions, like low degrees of freedom, as in a group analysis. P threshold FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#p-threshold) delves into more detail about this.

17.16.9 phantom

Any object you scan in an MRI machine that's intended only to help you calibrate your scanner. Phantoms can range from very simple (a tank of water) to very complicated (a plastic skull with a gelatin brain controlled by several motors to simulate head movements). The fact that they don't have brain responses is the key; you can use them to check your scanner or preprocessing paradigm, or introduce fake signal into a phantom scan and know that you won't be corrupted by real brain responses.

17.16.10 Plugin

In the context of Nipype, plugins are components that describe how a workflow should be executed. They allow seamless execution across many architectures and make the usage of parallel computation look so easy. For more see the introduction section (http://miykael.github.io/nipype-beginner-s-guide/nipype.html#execution-plugins) of this beginner's guide.

17.16.11 Positron Emission Tomography (PET)

An imaging method in which subjects are injected with a slightly radioactive tracer, and an extremely sophisticated and sensitive radition detector is used to localize increased areas of blood metabolism during some experimental task. PET offers better spatial resolution than *EEG (Electroencephalogram)*, but not as much as fMRI - on the order of tens of millimeters at best. Its temporal resolution is pretty poor, as well - within tens of seconds at best, making *block design* the only feasible design for PET studies. As well, PET scanners are very expensive, and so aren't around at many institutions. Nonetheless, studies have demonstrated one extremely useful aspect of PET - the ability to selectively label particular neurotransmitters, like dopamine, and hence get a chemically-specific picture of how one neurotransmitter is being used. SPM was originally developed for use with PET.

17.16.12 power

A statistical concept which quantifies the ability of your study to reliably detect an effect of a particular size. Studies with higher power can reliably detect smaller effects. A tremendous number of factors influence your study's power, from the ordering of your stimuli presentation to the noise characteristics of the scanner, but the one that's most under your control is your experimental design. High power is very desirable for fMRI studies, where effect sizes can often be extremely small, but it doesn't come without a cost; increasing the power of your study requires decreasing the *efficiency*, which can also be seen as assuming more information about the shape of your response. See Design FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#design) (and Jitter FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#jitter)) for tons more on power and efficiency and how to manipulate them both.

17.16.13 pre-whitening

A process by which signals that are corrupted by non-white noise - i.e., colored noise, or noise that is more prevalent at some frequencies than others - can be improved, by making the noise "whiter." This involves estimating the *autocorrelation* function of the noise, and then removing the parts of the noise that are influenced by previous noise values, leaving only independent or *white noise*. Whatever analysis is to be done on the signal is then carried out. Because this process makes "colored" noise into white noise, it's called whitening, and the "pre" part is because it happens before the model estimation (or other analysis) is done on the signal. This is a standard technique in many signal processing domains. See Basic Statistical Modeling FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#basic-statistical-modeling) for more details.

17.16.14 preprocessing

Any manipulation of your data done before you estimate your model. Usually this refers to a set of spatial transformations and manipulations like *realignment*, *normalization*, or *smoothing* done to decrease noise and increase signal strength. There are various preprocessing steps you can take in the temporal domain as well, like *temporal filtering* or *pre-whitening*. In SPM, "preprocessing" often refers to the specific set, in order, of slice timing correction, realignment, normalization and smoothing, which are grouped together in the interface and generally comprise the first steps of any analysis.

17.16.15 Principal Components Analysis (PCA)

A statistical technique for identifying components of your signal that explain the greatest amount of variance. In fMRI, it's used as a way of analyzing data that doesn't require a model or *design matrix*, but rather breaks the data down into a set of distinct components, which can be interpreted in some case as distinct sources of signal. These components can then (hopefully) be localized in space in some intelligible way. This enables you, theoretically, to discover what effects were "really" present in your experiment, rather than hypothesizing the existence of some effects and testing the significance of your hypothesis. It's been used more heavily in *EEG (Electroencephalogram)* research, but is beginning to be applied in fMRI, although not everything about the results it gives is well understood. Its use in *artifact* detection is clear, though. It differs from *Independent Components Analysis (ICA)*, an algorithm with similar goals, because the components it chooses explain the maximum amount of variance in the dataset, rather than maximizing the statistical independence of the components.

17.16.16 prospective motion correction

A form of *realignment* that is performed within the scanner, while the subject is actually being scanned. Rather than waiting until after the scan and trying to line up each functional image with the previous after the fact, prospective motion correction techniques aim to line up each functional image immediately after it is taken, before the next image is taken. Since TRs are typically on the order of a few seconds, these algorithms must operate very fast. Standard methods call for an extra RF pulse or two to be taken during one TR's pulse sequence, essentially to quantify how much the subject has moved during the TR. These algorithms can avoid some of the major problems of standard realignment algorithms, like biasing by activation and warping near susceptible regions. That extra functionality comes at the cost of time - it usually takes tens of milliseconds per TR to perform, which might mean taking one fewer slice or two.

17.16.17 psychophysiological interaction (PPI)

A term invented by Karl Friston (https://en.wikipedia.org/wiki/Karl_J._Friston) and the SPM group to describe a certain type of analysis for *functional connectivity*. They have argued that looking at simple correlations of signal between two regions may not be as interesting as looking at how those correlations change due to the experiment; i.e., does condition A induce a closer connection between two regions than condition B does? If so, these regions have a psychophysiological interaction (or PPI) - an interaction influenced both by psychological factors (the experimental condition) and physiological factors (the brain signal from another region). Check out Connectivity FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#connectivity) for more.

17.16.18 pulsatility

A type of *artifact* induced by the cardiac cycle. The beating of the heart pushes blood through the arteries and into the brain, and the rhythmic influx of blood actually causes small swellings and deflations in brain tissue, as well as other small movements, all timed to the heartbeat. As the heartbeat is often faster but around the same timescale as the TR, signal changes induced by cardiac movements can be unpredictable and difficult to quantify and remove. See Physiology and fMRI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#physiology-and-fmri) for more on physiological sources of artifacts.

17.16.19 pulse sequence

fMRI works by stimulating the brain with rapid magnetic pulses in an intense baseline magnetic field. The exact nature of those rapid pulses determines exactly what kind of fMRI signal you're going to get out. Many things about those pulses are standardized, but not all, and you can use different pulse sequences to take functional images, depending on your scanner characteristics and different parameters of your experiment. *Echo-planar Imaging (EPI)* and *spiral imaging* are two well-known functional pulse sequences; there are many others for other types of scans. Check out Scanning FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#scanning) and Physiology and fMRI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#physiology-and-fmri) for a little bit more.

17.16.20 Python

Python (https://www.python.org/) is a widely used general-purpose, high-level programming language. Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. Python becomes more and more the programming language for the scientific Neuroimaging field. This because the language is easy to learn and can be mastered by also none programmer in a rather short time. For more see Python's Wikipedia page (https://en.wikipedia.org/wiki/Python_%28programming_language%29).

17.17 Q

17.18 R

17.18.1 radiological convention

Radiological images (like fMRI) that are displayed where the left side of the image corresponds to the right side of the brain (and vice versa) are said to be in "radiological convention" or "radiological format." In radiological convention, left is right and right is left. Those crazy radiologists. This contrasts with *neurological convention*. Some image formats do not contain information saved as to what convention they're in, and Side Flipping can be an issue with those images. So be careful.

17.18.2 random-effects

An analysis that assumes that the subjects (or scanning sessions, or scanner runs, or whatever) you're drawing measurements from are randomly drawn from some distribution. The differences between them must thus be accounted for in accounting for the average effect size. This generally means evaluating effects within each subject (session/run/etc.) separately, to allow for the possibility of differential responses, which means separate design matrices and estimations. This costs you a significant amount of *power* from a fixed-effects analysis, because you only end up having as many degrees of freedom in your test as you have subjects (sessions/runs/etc.), which is generally far smaller than the number of measurements (i.e., functional images). The advantage is a gain in inferential power: a random-effects analysis allows you to make inferences about the population from which the subjects were drawn, not just the subjects themselves. Fixed-effects analyses of any kind do not allow this type of inference. The analyses generally done in neuroimaging programs is technically a *mixed-effects* analysis, because they include both fixed and random effects. Check out Random and Fixed Effects FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#random-and-fixed-effects) for more.

17.18.3 rapid event-related designs

Any *event-related design* in which trials occur too fast for the *Hemodynamic Response Function (HRF)* to return to baseline in between trials. This generally corresponds to an *Inter-stimulus Interval (ISI)* of less than 20-30 seconds or so. These designs contrast with *long event-related designs*. They are more difficult to analyze than long event-related designs, because you have to make assumptions about the way that the hemodynamic response to different events adds up. They compensate for this difficulty by being having much more *power* and *efficiency* than long event-related designs - *so long* as the mean ISI in the design is properly varied or *jittered*. This gain comes from the increased number of trials per unit time, but necessitates proper jitter. See Design FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#design) for more, and Jitter FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#jitter) for a good deal about rapid designs specifically.

17.18.4 realignment

Also called motion correction. A spatial preprocessing step in which functional images are lined up together, so a single voxel in the grid corresponds to the same anatomical location during the whole experiment. This step is needed due to subtle head motions from the subjects; even with a bite bar or head mount, subjects move their head slightly during an experiment, and so the functional images that are taken end up being slightly out of register with each other. Realignment aims to line them back up again. See Realignment FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#realignment) for much much more.

17.18.5 reference slice

A term used in *slice timing* correction to denote the slice of the brain that no correction is done on. All other slices of each functional image will have their voxels' timecourses slightly shifted in the temporal domain so that they take on the values they "would have had" if the whole brain had been sampled at the same moment as the reference slice. See Slice Timing FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#slice-timing) for more, and for how to choose a reference slice.

17.18.6 Region of Interest (ROI)

Any subset of *Voxels* within the brain that you want to investigate further. They might comprise an anatomical structure, or a cluster of activated voxels during your task. A ROI needn't be spatially contiguous, although they often are. Subtypes are *anatomical ROI* and *functional ROI*. They can be identified before or after a standard *general linear model (GLM)* analysis, and they often represent some area of pre-existing theoretical interest. They're often saved as either lists of coordinates (all coordinates in the list make up the ROI) or image masks, a special type of image file where every voxel in the ROI has intensity 1 and every voxel not in the ROI has intensity 0. Several further analyses can be performed once you've identified some regions of interest. See ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) for some thoughts on them.

17.18.7 render, rendering

A three-dimensional object like the brain can be difficult to visualize in a two-dimensional picture. Several graphics packages provide facilities to make a three-dimensional picture of the brain that shows the folds of the surface, and often allows zooming and rotation of the whole 3-D object. This process of making a 3-D image is called rendering. All the major neuroimaging software packages provide some rendering package. They all allow you to superimpose patterns of activation on those 3-D objects, to allow a better visualization of the 3-D nature of the activations. Rendering is often connected with other 3-D visualization methods, like *inflation* or *flattening*.

17.18.8 reverse / inverse normalization

After *normalization*, you have some set of transformation parameters which specify how the individual subject's brain was warped and shifted to match the standard template brain. One thing you could do at that point would be to identify some *functional ROI* in the normalized group results, or some *anatomical ROI* on a standard brain like the MNI template or Talairach brain. Reverse normalization would entail, then, inverting the transformation matrix of normalization and applying the reversed matrix to some anatomical or functional ROI made at the normalized, standard brain level. This reverse-normalized ROI would then be warped to fit your individual subject's brain, and you could then analyze any non-normalized images you had of theirs with it. Given that normalization induces some interpolation errors and localization problems into your images, this might be a great way to save labor on hand-drawing ROIs but still look at non-normalized results. See ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) for more info on why you'd want to analyze data at the individual level, and Normalization FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#normalization) for more about the normalization process.

17.18.9 run

A term used to describe a single pass-through of a given experimental paradigm, which generally corresponds with a single chunk of time between turning the scanner on and turning it off. A given experiment for one subject often consists of several runs, which are often all modeled together in a *fixed-effects* analysis. Generally, it does not mean the whole time a subject is in the scanner if there are several chunks of scanning time in there. Often used interchangeable (and confusingly) with *session*.

17.19 S

17.19.1 scanner drift

See *drift*.

17.19.2 script

in *MATLAB*, a type of .m file that doesn't take arguments or give output, but merely operates in the base workspace. Essentially scripts are just a text file containing a bunch of Matlab commands exactly as if you'd typed them, in order, at the Matlab prompt when you ran the script. Scripts are contrasted with functions, which have their own workspaces and don't have access to the base workspace. Most SPM sub-programs are functions, but not all of them.

17.19.3 segmentation

A spatial step in which an automated algorithm classifies a brain image into different tissue types. Standard segmentation programs start with an MRI image - generally, but not always, an anatomical scan - and give out images of all the gray matter in the brain, all the white matter, and all the cerebrospinal fluid (CSF). Each voxel is thus labeled uniquely as being one of the three standard tissue types. Those images can then be used to make mask images (to restrict analysis to gray matter only, for example) or to do *Voxel-based Morphometry (VBM)*, or a lot of other things. Segmentation can be pretty inexact, due to problems like *partial voluming* and other issues, so advanced segmentation algorithms these days sometimes do a "soft classification," where voxels are labeled only with a probability of being a certain tissue type, rather than a definite label. Other segmentation algorithms go farther and use anatomical information to classify voxels into different structures as well as different tissue types. See Segmentation FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#segmentation) for lots more.

17.19.4 session

An ambiguous term usually used to denote the exact same thing as *run*: the chunk of time in an experiment between turning the scanner on and turning it off, during which you have one pass of your experimental paradigm. Oftentimes, the experiment on one subject will have several sessions, which might all be the same paradigm or different ones. Unfortunately, this term has also been used to denote the whole single-subject experiment; i.e., one scanning session is the whole time you have the person in the scanner, which might include several different runs.

17.19.5 Signal-to-Noise Ratio (SNR)

One of the most self-explanatory terms out there. If you can quantify the amount of signal you have in a measurement and the amount of noise, then you divide the former by the latter to get a ratio - specifically, your signal-to-noise ratio, or SNR. Your SNR is a far more valuable measure of how much *power* your measurement will have than, say, average intensity; if the measurement is brighter, that could mean more signal or more noise. Things like *smoothing* change average intensity unpredictably, but always aim to increase SNR. Calculating SNR can be tricky, because it requires some determination (or at least estimation) of how much noise your measurement has, which may not be known. But things like *phantom* measurements can help. See Scanning FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#scanning) for a little bit of commentary on how your scanning parameters can tweak your SNR.

17.19.6 single-subject canonical

An image distributed with *SPM (Statistical Parametric Mappin)* that is a very clear anatomical scan of a single brain (as opposed to the average scan of many brains, which is how brain templates like the MNI brain are made). The single-subject canonical is often used as a background to superimpose normalized results onto, because the brain is roughly average in shape and more or less lines up with the MNI template. It's also a very, very clear scan (made by averaging many scans of the same brain together) and so is much clearer than a standard in-plane anatomical scan for a single subject might be. However, the single-subject canonical is not an exact map onto the MNI or Talairach templates; activation which appears to be in one structure on the canonical image may not lie in that structure in either template brain. This image is generally found in the SPM directory, in the /canonical subdirectory.

17.19.7 slice timing

A spatial *preprocessing* step which aims to correct for the fact that not all slices of a functional volume are sampled at the same instant. Functional images aren't acquired instantly - they are sampled across the whole TR, so with a descending *pulse sequence* and a 2-second TR, the bottom of the brain is sampled almost two seconds after the two of the brain. If every voxel in the brain is analyzed with exactly the same model, then the onsets you've specified are going to be correct for some parts of the brain and wrong for others. If you say a trial happens at time 1, in the above example, and the TR starts right then, your onset is almost 2 seconds off for voxels at the bottom of the brain, because by the time you sample them, they're 2 seconds into their hemodynamic response already. Slice timing correction aims to fix this problem by simply time-shifting or interpolating all the voxels in the brain to line up with a *reference slice*. The methods for doing this are fairly uncontroversial and generally accepted as necessary for all *event-related design*. See Slice Timing FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#slice-timing) for more.

17.19.8 slice thickness

Sometimes when you take a functional MRI sequence, your *Voxels* aren't *isotropic* - there is a given matrix within a slice (often 64x64 voxels), and a certain set of slices (usually ranging from a few to a few dozen). Your slice thickness is exactly what it sounds like - how thick, in millimeters, your slices are. This is also called the through-plane resolution of your voxels - voxels are often thicker between slices than within a slice. Sometimes you'll leave a gap between slices; this is called the "skip" distance and isn't factored into your slice thickness.

17.19.9 small-volume correction (SVC)

If you have a pre-existing hypothesis about a particular region in the brain - an anatomical or functional ROI from another study, say - then you might want to search within only that region for activation. This helps avoid the multiple-comparison problem for thresholding; instead of correcting your threshold for the tens of thousands of voxels in the whole brain, you can say you're only looking within a small region and correct for only the hundreds or thousands of tests within a much smaller region. This is called small-volume correction. It's available in SPM through the results interface's S.V.C. button. This button is also used sometimes to merely save a cluster or region as a functional ROI in SPM, rather than actually looking at the corrected statistics. See P threshold FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#p-threshold) for more on thresholding.

17.19.10 smoothing

A spatial *preprocessing* step in which your functional images are blurred slightly. Each voxel's intensity is replaced with a weighted average of its own intensity and some voxels around it; this is accomplished by convolving a Gaussian function - the *Smoothing kernel* - with the intensity at each voxel. The amount of blurring is determined by the size of the kernel. Smoothing can greatly increase your *Signal-to-Noise Ratio (SNR)*, as well as increase the chance of getting group activations (by increasing the size and hence overlap of functional regions) and validating the assumptions of *Gaussian random field* theory if you're doing that sort of *Family-wise error correction (FWE)*. The downside of smoothing is, well, it makes your data blurrier. This is a problem if you're trying to decide whether one voxel or its neighbor is active, or if you're worried about smearing activation across anatomical or functional boundaries in the brain. It effectively reduces the resolution of your images. Smoothing FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#smoothing) has tons more on why to smooth and why not to smooth.

17.19.11 smoothing kernel

A generally Gaussian function which is convolved with voxel intensities in a given functional image during *smoothing*. The "size" of the kernal is the FWHM (full-width half-maximum) measurement of the Gaussian function. Common kernel sizes for fMRI range between 2 and 12 mm, depending on what you're looking for. See Smoothing FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#smoothing) for more on choosing a kernel size.

17.19.12 spatial frequency

Like any other signal, images can be analyzed in terms of their frequency. A gross simplification might be looking at the image intensities of neighboring voxels as a timecourse, and finding the frequencies of the waveforms contained within. In real life, finding spatial frequency is a little trickier, but the idea is the same. Low spatial frequency equals slow change in intensity; areas with low spatial frequency in an image are largely homogeneous, smooth, and less-varying. High spatial frequency equals fast change in intensity; areas of high spatial frequency in an image are often edges, or choppy patterns. *k-space* is a way to view images in terms of their spatial frequency.

17.19.13 spatial preprocessing

See *preprocessing*; this term refers specifically to spatial transformations done before analysis, like *normalization*, *smoothing*, *slice timing* correction or *realignment*, and excluding temporal manipulations like *high-pass filter* or *pre-whitening*.

17.19.14 spatial smoothing

A measure of *spatial frequency*. Spatial smoothness just measure the amount of low-spatial-frequency information in an image or a local region of an image. This is a way of quantifying how smoothly an image varies across the whole volume or a small chunk of it. Images have to have a relatively high spatial smoothness to satisfy the assumptions of *Gaussian random field* theory and be eligible for Gaussian-random-field *Family-wise error correction (FWE)*. Increasing their spatial smoothness can be accomplished with, of all things, *smoothing*. Crazy. See Smoothing FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#smoothing) and P threshold FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#p-threshold) for the relationship between smoothness and thresholding.

17.19.15 spiral imaging

A particular *pulse sequence* in which *k-space* is sampled in a spiraling trajectory, rather than in discrete lines. Spiral imaging avoids some of the common *artifact* than can plague other sequences like *Echo-planar Imaging (EPI)*: geometric distortions, ghosting, or radical displacement. Spiral artifacts tend to be simply blurring of greater and lesser degree. Some spiral sequences can be more susceptible to *dropout*, but spiral in-out sequences seem to recover a great deal of signal from all parts of the brain. See Scanning FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#scanning) for a bit more on spiral sequences.

17.19.16 spiral-in, spiral-out, spiral in-out, spiralio

Different variations of spiral pulse sequences. In spiral-in, *k-space* is sampled in an inward-spiraling trajectory during the TR; in spiral-out, *k-space* is sampled in an outward-spiraling trajectory. Spiral in-out (also called spiralio) sequences do both, sampling k-space on an inwards spiral followed by an outwards spiral during the same TR and averaging the two images together. Spiral in-out sequences in particular do an excellent job at avoiding *dropout* in many areas of the brain traditionally thought to be difficult to image due to dropout. Check out Scanning FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#scanning) for more.

17.19.17 SPM (Statistical Parametric Mappin)

A software package for neuroimaging analysis, written in *MATLAB* and distributed freely. Probably one of the most widely-used package worldwide, currently. Has an easy-to-learn interface combined with some of the most sophisticated statistical modeling available. See SPM in a Nutshell FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#spm-in-a-nutshell) for a more detailed summary of what SPM is. For everything else, see SPM's main homepage (http://www.fil.ion.ucl.ac.uk/spm/) and the SPM Mailinglist (https://www.jiscmail.ac.uk/cgi-bin/webadmin?A0=spm).

17.19.18 stimulus-correlated motion (SCM)

Head motion during an experiment is a big enough problem to start with. But random head motion can be dealt with by *realignment* and including your motion parametes in your design matrix, to eliminate any signal correlated with head motion. So why doesn't everyone just do that? Because if your subject moved their head in correlation with your task paradigm, removing motion-correlated signal will also remove task-correlated signal - which is what you're looking for. So stimulus-correlated motion is a big problem because it prevents you from regressing out motion-related activity. Evaluating your SCM should be a priority for anyone who includes motion parameters in their design matrix, particularly if you don't use a bite bar or if you have an emotionally-intense paradigm. Check out Realignment FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#realignment) for more.

17.19.19 structural equation modeling (SEM)

A statistical method for analyzing *functional connectivity*. Structural equation modeling (SEM) allows you to start with a set of *Region of Interest (ROI)* and figure out what the connection strengths between them are, via a model-fitting process. It can't be used to determine the directionality of connection, but it can do a good job describing which connections are strong and which are weak, which can be crucial in ruling out certain theoretical constructs. See Connectivity FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#connectivity) for lots more on this.

17.19.20 surface mapping

The cortex, where much of the brain's processing takes place, could be flattened out into a flat sheet. But in the head, it's all crinkled up into sulci and gyri. If you ignore the folds of the brain and simply analyze it like it's all one homogenous shape - as traditional voxel-based analysis do - then you may well miss important principles of how activation is organized, and you might even miss real activations in general. Surface mapping techniques are related to *inflation* and *flattening* techniques, and surface mapping is in fact a necessary prerequisite for those. Surface mapping simply starts with a high-quality anatomical scan, and builds a three-dimensional model of the folds and curves of the brain, which is then linked to particular voxels in the functional analysis. This allows the activation from the functional images to be mapped not just to a particular voxel, but to a particular point on the surface of the cortex. This surface can then be manipulated and visualized in far more interesting ways than simple voxel-based pictures allow.

17.19.21 susceptibility

Also called magnetic susceptibility. Used to describe regions where magnetic fields are generally more distorted, chopped up, and subject to *dropout*, due to the tissue characteristics of a region. Usually, regions of high susceptibility (try typing that five times fast) are near tissue/air interfaces, or interfaces between two different types of tissue, where the magnetic differences between the two materials causes distortions in the local field. High-susceptibility regions traditionally include the orbitofrontal cortex, medial temporal lobe, and many subcortical structures. Spiral in-out imaging has shown good promise at dealing with susceptibility-induced dropout. See Scanning FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#scanning) for more.

17.20 T

17.20.1 tal2mni

A script written by Matthew Brett (check the internet for tal2mni.m or mni2tal.m), which aims to convert a set of XYZ coordinates from a given point in the *Talairach* atlas brain into the same anatomical point in the Montreal Neurological Institute (MNI) standard template brain. The *Talairach* brain, which is used as the *normalization* template for AFNI, BrainVoyager, and other programs, differs slightly from the MNI brain in several ways, particularly in the inferior parts of the brain. In order to use facilities like the *Talairach Daemon* or other Talairach-coordinate lookups to make ROIs for normalized SPM results, or in order to report *Talairach* data in MNI coordinates, it's necessary to convert the *Talairach* coordinates into MNI space with this script. It's not a perfect mapping, but it's widely used. See ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) and Normalization FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#normalization) for more.

17.20.2 Talairach

In 1988, Talairach and Tournoux published a widely-cited paper created a common reference coordinate system for use in the human brain. The paper set forth axes labels and directions, an origin at the anterior commissure, and anatomical and cytoarchitectonic labeling for many individual coordinate points within the brain. The coordinate system is based on one reference brain they dissected, sometimes referred to as the Talairach brain. The coordinate system has been widely adopted, and many algorithms have sprung up to normalize arbitrary brains to the Talairach reference shape. Coordinates in the reference system are said to be in Talairach space, and the full listing of coordinates and their anatomical locations is called the Talairach atlas. (Tournoux pretty much got the short end of this whole stick.) Although the coordinate system has been widely used and has proven very valuable for standing reporting of results, it has drawbacks: the Talairach brain itself is a fairly unrepresentative single subject (and differs significantly from a more average template brain - see *tal2mni*), it ignores left-right hemispheric differences as only one hemisphere was labeled, and there are no MRI pictures available of it to be directly comparable. Some programs, like *SPM (Statistical Parametric Mappin)*, have avoided using the Talairach brain for normalization, but Talairach labeling is pretty much inescapable at this point. Check out ROI FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#roi) for a little more on all this, as well as Normalization FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#normalization).

17.20.3 Talairach Daemon

A very nice software package hosted by UT-San Antonio and developed by Lancaster et. al, the Talairach Daemon takes in a set of coordinates in *Talairach* space and spits out a set of anatomical labels for each point - hemisphere, anatomical area, brodmann area, tissue type - based on the *Talairach* atlas. This allows you, in an automated fashion, to label your results in a common space with many other researchers.

17.20.4 task-correlated motion

See *stimulus-correlated motion (SCM)*.

17.20.5 temporal derivative

Derivative of a function with respect to time. In SPM, the temporal derivative of the *canonical HRF* looks something like the canonical but can be used as a *basis function*, to model a degree of uncertainty as to the exact onset of the HRF. See HRF FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#hrf) for more.

17.20.6 temporal filtering

A filter applied in the temporal domain to some signal to help cut noise. Temporal filters knock out some frequencies in a given signal while allowing others to pass through; some types include *high-pass filter*, *low-pass filter*, and *band-pass filter*. In fMRI, applying some temporal filtering is a terrifically good idea, because noise is heavily concentrated in some parts of the frequency spectrum. See Temporal Filtering FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#temporal-filtering) for more.

17.20.7 Tesla

The Tesla is the standard international (or metric system, if you must) unit of magnetic flux density. It's abbreviated simply as T. It measures, in a nutshell, the strength of a standing magnetic field at a given point. It's named after Nikola Tesla, the engineer who discovered the rotating magnetic field back in the 19th century. The strength of an MRI scanner is measured in Tesla. Most scanners in current use for humans are rated as 3T; human scanners up to 7T can be found around. For comparison, the Earth's standing magnetic field is around 2.5 * 10e-5 T.

17.20.8 time-locked averaging

A technique in signal processing for signals where some repeating signal is corrupted by random noise. If you know the timepoints in the timeseries when the signal starts and can choose a window of time following the start to look at - say, 30 seconds - then you could take the 30-second chunk following each signal onset and average all those 30-second chunks together. If you have five signal onsets, then you have 5 windows; you average the first timepoint in each window together (all 5 of them), then the second timepoint in each window together (all 5 of those), then the third timepoint, etc. This creates an average time window - the average response following a signal onset. If the noise is roughly random, it should average to zero, and you'll get a clearer picture of your signal than from any individual response. The resulting *peristimulus timecourse* is called "time-locked" because it always describes a given time following the onsets - it's locked in time to the condition's onsets. This technique has long been used in EEG, and with the advent of a *event-related design*, it began to be used in fMRI as well. The technique may not be appropriate for *rapid event-related designs*; when the window following an onset overlaps the onset of other signals, the final timecourse can be muddled by other signals.

17.20.9 timecourse, timeseries

A list of numbers that are taken to represented some measurement sampled over time. Each point in the timeseries represents a specific point in time; neighboring points represent neighboring moments, later points represent later points in time, etc. Many scientific domains deal with timeseries data, and so a good deal of research has been done on how to deal with any peculiar characteristics they might have - *autocorrelation*, etc. In fMRI, the most common timeseries would be the series of measurements from a specific voxel across all the functional images - that repeated measurement represents a series of samples over time in (we hope) one unique point in the brain.

17.21 U

17.21.1 unwarping

A *preprocessing* technique which attempts to eliminate some of the residual effects of head motion after realignment. The method attempts to estimate a map of the inhomogeneities in the magnetic field and thus map regions of high *susceptibility*, and calculate how those regions might have distorted the data around them, once the head motion parameters are known. Unfortunately, this method is currently (and probably will always be) available only for *Echo-planar Imaging (EPI)* functional data, not spiral, due to the models of geometric distortion used. See Realignment FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#realignment) for a bit more.

17.22 V

17.22.1 variable ISI

Stands for variable inter-stimulus interval. A type of experiment in which a varying amount of time separates the beginning of all stimuli - trials can be all the same length or all different, but the onsets of stimuli aren't all the same length of time apart. variable ISI studies most often have ISIs that are randomized within certain extremes, not just arbitrarily variable. Generally only a *event-related design* are variable ISI, although there's no reason why you couldn't have a limited-variability-ISI *block design* experiment. variable ISI event-related experiments, though, are much better than *fixed ISI* experiments at both *efficiency* and *power*, especially as the ISI increases. In general, several empirical studies have shown that for event-related designs, *variable ISI* is the way to go. For block designs, the difference is fairly insignificant, and variable ISI can make the design less powerful, depending on how it's used. See Jitter FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#jitter) for more on the difference between fixed and variable.

17.22.2 voxels

One "dot" in a 3-D picture. Like "pixel" for 2-D pictures, but it's three-dimensional. Voxels have a given size, usually a few millimeters in any direction (although they can be *isotropic* or *anisotropic*). Their size is specified in millimeters generally, like 2x2x3.5; the third dimension is generally the through-plane size or *slice thickness*. In a given brain, you'll often have tens of thousands of voxels, even if you haven't resampled your voxels to be smaller during *preprocessing*. Voxels are specified on a coordinate system that's different than the millimeter coordinate system; millimeters coordinates have their origin in the middle of the image (and so can be negative), whereas voxel coordinates start counting in one corner of the image and are always positive.

17.22.3 Voxel-based Morphometry (VBM)

A type of analysis which doesn't look at functional images, but instead looks at the differences between subjects' anatomy. Anatomical images are segmented into different tissue types, and the measurements generally looked at are the total volume of gray matter (or white matter or CSF) in a given anatomical structure. This type of analysis is about the form, or morphometry of the brain, and it's based not on the surface of the brain or any dissection of it but on arbitrarily-sampled *Voxels* - hence the name. See Segmentation FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#segmentation) for more.

17.23 W

17.23.1 white noise

Noise which is random and independent from measurement to measurement. In other words, it is equally strong in all frequencies. White noise is nice because it tends to average to zero, which enables the use of many simple *smoothing* techniques to get rid of it, although it tends to defeat filtering techniques. It's in contrast to colored noise, which has some correlation from timepoint to timepoint. fMRI noise tends to be pretty white in the spatial domain (with some exceptions) and severely colored in the temporal domain. See Smoothing FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#smoothing) and Temporal Filtering FAQ (http://miykael.github.io/nipype-beginner-s-guide/faq.html#temporal-filtering) for a little more.

17.23.2 whitening

See *pre-whitening*.

17.23.3 Workflow

Workflows are the core elements of Nipype and can also be called pipelines. Workflows consists of Nodes, MapNodes and other Workflows and define the sequential order data processing and other algorithms should be executed by. For more see this beginner's guide introduction section (http://miykael.github.io/nipype-beginner-s-guide/nipype.html#workflow-engine).

17.24 X

17.25 Y

17.26 Z

17.27 Numbers

17.27.1 1.5T, 3T, 4T, 7T (etc.)

Ratings of different strengths of MRI scanners. T is the abbreviation for *Tesla*, the international standard unit for magnetic flux density.

CHAPTER

EIGHTEEN

DOWNLOADS

Download Nipype here: Nipype Homepage (http://nipype.readthedocs.io/en/latest/users/install.html).

Download this beginner's guide as a PDF here: Nipype Beginner's Guide (http://github.com/miykael/nipype-beginner-s-guide/blob/master/NipypeBeginnersGuide.pdf?raw=true).

Download all scripts from this beginner's guide here:Scripts (http://github.com/miykael/nipype-beginner-s-guide/blob/master/scripts).

Download the dataset DS102: Flanker task (event-related) (https://openfmri.org/dataset/ds000102) used as the tutorial dataset for this beginner's guide directly here: ds102_raw.tgz (http://openfmri.s3.amazonaws.com/tarballs/ds102_raw.tgz).