Neuroimaging in Python Documentation

Release 0.1

The Neuroimaging in Python documentation team.

July 07, 2009

CONTENTS

Part I

User Guide

CHAPTER

ONE

INTRODUCTION

As you can see, we do not yet have much of a user guide for NIPY. We are spending all our effort in developing the building blocks of the code, and we have not yet returned to a guide to how to use it.

We are starting to write general *Tutorials*, that include introductions to how to use NIPY code to run analyses.

1.1 What is NIPY for?

The purpose of NIPY is to make it easier to do better brain imaging research. We believe that neuroscience ideas and analysis ideas develop together. Good ideas come from understanding; understanding comes from clarity, and clarity must come from well-designed teaching materials and well-designed software. The software must be designed as a natural extension of the underlying ideas.

We aim to build software that is:

- clearly written
- clearly explained
- a good fit for the underlying ideas
- a natural home for collaboration

We hope that, if we fail to do this, you will let us know. We will try and make it better.

The NIPY team

1.2 A history of NIPY

Sometime around 2002, Jonthan Taylor started writing BrainSTAT, a Python version of Keith Worsley's FmriSTAT package.

In 2004, Jarrod Millman and Matthew Brett decided that they wanted to write a grant to build a new neuoimaging analysis package in Python. Soon afterwards, they found that Jonathan had already started, and merged efforts. At first we called this project *BrainPy*. Later we changed the name to NIPY.

In 2005, Jarrod, Matthew and Jonathan, along with Mark D'Esposito, Fernando Perez, John Hunter, Jean-Baptiste Poline, and Tom Nichols, submitted the first NIPY grant to the NIH. It was not successful.

In 2006, Jarrod and Mark submitted a second grant, based on the first. The NIH gave us 3 years of funding for two programmers. We hired two programmers in 2007 - Christopher Burns and Tom Waite - and began work on refactoring the code.

Meanwhile, the team at Neurospin, Paris, started to refactor their FFF code to work better with python and NIPY. This work was by Alexis Roche, Bertrand Thirion, and Benjamin Thyreau, with some help and advice from Fernando Perez.

In 2008, Fernando Perez and Matthew Brett started work full-time at the UC Berkeley Brain Imaging Center. Matthew in particular came to work on NIPY.

CHAPTER

TWO

DOWNLOAD AND INSTALL

This page covers the necessary steps to install and run NIPY. Below is a list of required dependencies, along with additional software recommendations.

NIPY is currently *ALPHA* quality, but is rapidly improving. If you are trying to get some work done wait until we have a stable release. For now, the code will primarily be of interest to developers.

2.1 Dependencies

2.1.1 Must Have

Python 2.4 or later

NumPy 1.2 or later

SciPy 0.7 or later Numpy and Scipy are high-level, optimized scientific computing libraries.

PyNifti We are using pynifti for the underlying file IO for nifti files.

gcc NIPY does contain a few C extensions for optimized routines. Therefore, you must have a compiler to build from source. XCode (OSX) and MinGW (Windows) both include gcc. (*Once we have binary packages, this requirement will not be necessary.*)

2.1.2 Strong Recommendations

iPython Interactive python environment.

Matplotlib 2D python plotting library.

2.2 Installing from binary packages

Currently we do not have binary packages. Until we do, the easiest installation method is to download the source tarball and follow the *Building from source code* instructions below.

2.3 Building from source code

Developers should look through the *development quickstart* documentation. There you will find information on building NIPY, the required software packages and our developer guidelines.

If you are primarily interested in using NIPY, download the source tarball and follow these instructions for building. The installation process is similar to other Python packages so it will be familiar if you have Python experience.

Unpack the tarball and change into the source directory. Once in the source directory, you can build the neuroimaging package using:

```
python setup.py build
```

To install, simply do:

```
sudo python setup.py install
```

Note: As with any Python installation, this will install the modules in your system Python *site-packages* directory (which is why you need *sudo*). Many of us prefer to install development packages in a local directory so as to leave the system python alone. This is mearly a preference, nothing will go wrong if you install using the *sudo* method. To install in a local directory, use the **–prefix** option. For example, if you created a `local` directory in your home directory, you would install nipy like this:

```
python setup.py install --prefix=$HOME/local
```

CHAPTER

THREE

TUTORIALS

3.1 Basic Data IO

Accessing images using nipy:

Nifti is the primary file format

3.1.1 Load Image from File

```
from nipy.core.api import load_image
infile = 'myimage.nii'
myimg = load_image(infile)
```

3.1.2 Access Data into an Array

This allows user to access data in a numpy array.

Note: This is the correct way to access the data as it applies the proper intensity scaling to the image as defined in the header

```
from nipy.core.api import load_image
import numpy as np
myimg = load_file('myfile')
mydata = np.asarray(myimg)
mydata.shape
```

3.1.3 Save image to a File

```
from nipy.core.api import load_image,save_image
import numpy as np
myimg = load_file('myfile.nii')
newimg = save_file(myimg,'newmyfile.nii')
```

3.1.4 Create Image from an Array

This will have a generic CoordinateMap with Unit step sizes

```
from nipy.core.api import fromarray, save_image
import numpy as np
rawarray = np.zeros(43,128,128)
innames='kij'
outnames='zyx'
newimg = fromarray(rawarray, innames, outnames)
```

Images have a Coordinate Map.

The Coordinate Map contains information defining the input and output Coordinate Systems of the image, and the mapping between the two Coordinate systems.

Basics of the Coordinate Map

Here is an examples file image_fromarray.py that shows the use of io and Coordinate Maps.

```
"""Create a nifti image from a numpy array and an affine transform."""

import os

import numpy as np

from nipy.core.api import fromarray, Affine
from nipy.io.api import save_image, load_image
from nipy.utils.tests.data import repository

# Load an image to get the array and affine
filename = str(repository._fullpath('avg152T1.nii.gz'))
assert os.path.exists(filename)

# Use one of our test files to get an array and affine (as numpy array) from.
img = load_image(filename)
arr = np.asarray(img)
affine_array = img.coordmap.affine.copy()

##############################################################################
# START HERE
##############################################################################


# 1) Create a CoordinateMap from the affine transform which specifies
# the mapping from input to output coordinates.

# Specify the axis order of the input coordinates
input_coords = ['k', 'j', 'i']
output_coords = ['z','y','x']
#or
innames = ('kji')
outnames = ('zyx')
# either way works

# Build a CoordinateMap to create the image with
affine_coordmap = Affine.from_params(innames, outnames, affine_array)

# 2) Create a nipy image from the array and CoordinateMap

# Create new image
newimg = fromarray(arr, innames=innames, outnames=outnames,
```

```
                                                coordmap=affine_coordmap)

##############################################################################
# END HERE, for testing purposes only.
##############################################################################
# Imports used just for development and testing.  User's typically
# would not uses these when creating an image.
from tempfile import mkstemp
from nipy.testing import assert_equal

# We use a temporary file for this example so as to not create junk
# files in the nipy directory.
fd, name = mkstemp(suffix='.nii.gz')
tmpfile = open(name)

# Save the nipy image to the specified filename
save_image(newimg, tmpfile.name)


# Reload and verify the affine was saved correctly.
tmpimg = load_image(tmpfile.name)
assert_equal(tmpimg.affine, affine_coordmap.affine)
assert_equal(np.mean(tmpimg), np.mean(img))
assert_equal(np.std(tmpimg), np.std(img))
assert_equal(np.asarray(tmpimg), np.asarray(img))

# cleanup our tempfile
tmpfile.close()
os.unlink(name)
```

3.2 Basics of the Coordinate Map

When you load an image it will have an associated Coordinate Map

```
from nipy.core.api import load_image
infile = 'Talairach-labels-1mm.nii'
myimg = load_image(infile)
coordmap = myimg.coordmap
```

The Coordinate Map contains information defining the input and output Coordinate Systems of the image, and the mapping between the two Coordinate systems.

For more on Coordinate Systems, Coordinates and thier properties `neuroimaging.core.reference.coordinate_system`

You can introspect a coordinate map

```
In [5]: coordmap.input_coords
Out[5]: {'axes': [('k', <Coordinate:"k", dtype=[('k', '<f8')]>),
('j', <Coordinate:"j", dtype=[('j', '<f8')]>),
('i', <Coordinate:"i", dtype=[('i', '<f8')]>)],
 'name': 'input-reordered'}
In [7]: coordmap.input_coords.axisnames
Out[7]: ['k', 'j', 'i']

In [8]: coordmap.output_coords
```

```
Out[8]: {'axes': [('z', <Coordinate:"z", dtype=[('z', '<f8')]>),
('y', <Coordinate:"y", dtype=[('y', '<f8')]>),
('x', <Coordinate:"x", dtype=[('x', '<f8')]>)],
'name': 'output-reordered'}
In [9]: coordmap.output_coords.axisnames
Out[9]: ['z', 'y', 'x']
```

Note: People using matlab are used to seeing the Coordinate System (i,j,k) mapping to the Coordinate System (x,y,z). This is due to fortran ordered reading of the data.

Numpy's default ordering is C ordered which is why in this case (k,j,i) maps to (z,y,x)

A Coordinate Map has a mapping from the *input* Coordinate System to the *output* Coordinate System

Here we can see we have a voxel to millimeter mapping from the voxel space (k,j,i) to the millimeter space (z,y,x)

We also know from the dtype attribute that the Axes in this Coordinate System are of type '<f8' *little-endian float64*

We can also get the name of the respective Coordinate Systems that our Coordinate Map maps between

```
In [20]: coordmap.input_coords.name
Out[20]: 'input-reordered'

In [19]: coordmap.output_coords.name
Out[19]: 'output-reordered'
```

A Coordinate Map is two Coordinate Systems with a mapping between them. Formally the mapping is a function that takes points from the input Coordinate System and returns points from the output Coordinate System.

Often this is simple as applying an Affine transform. In that case the Coordinate System may well have an affine property which returns the affine matrix corresponding to the transform.

```
In [11]: coordmap.affine
Out[11]:
array([[   1.,    0.,    0.,  -72.],
       [   0.,    1.,    0., -126.],
       [   0.,    0.,   -1.,   90.],
       [   0.,    0.,    0.,    1.]])
```

If you call the Coordinate Map you will apply the mapping function between the two Coordinate Systems. In this case from (k,j,i) to (z,y,x)

```
In [13]: coordmap([1,2,3])
Out[13]: array([ -71., -124.,   87.])
```

It can also be used to get the inverse mapping, or in this example from (z,y,x) back to (k,j,i)

```
In [14]: coordmap.inverse_mapping([-71.,-124.,87.])
Out[14]: array([ 1.,  2.,  3.])
```

We can see how this works if we just apply the affine ourselves. Notice the affine is using homogeneous coordinates so we need to add a 1 to our input. (And note how a direct call to the coordinate map does this work for you)

```
In [15]: coordmap.affine
Out[15]:
array([[   1.,    0.,    0.,  -72.],
       [   0.,    1.,    0., -126.],
       [   0.,    0.,   -1.,   90.],
```

```
       [   0.,    0.,    0.,    1.]])

In [17]: import numpy as np

In [18]: np.dot(coordmap.affine, np.transpose([1,2,3,1]))
Out[18]: array([ -71., -124.,   87.,    1.])
```

Note: The answer is the same as above (except for the added 1)

3.3 Specifying a GLM in NiPy

In this tutorial we will discuss NiPy's specification of a typical event-related fMRI model.

This involves:

- experimental model: a description of the experimental protocol (function of experimental time)
- neuronal model: a model of how a particular neuron responds to the experimental protocol (function of the experimental model)
- hemodynamic model: a model of the BOLD signal at a particular voxel, (function of the neuronal model)

3.3.1 Experimental model

Categorical designs

This design is the canonical "faces" vs. "objects" type design. For an event-related design, we can model the experiment as

$$E = \sum_{j=1}^{10} \delta_{(t_j, a_j)} \tag{3.1}$$

Formally, this can be thought of as realization of a *marked point process* that says we observe 10 points in the space $\mathbb{R} \times E$ where *E* is the set of all event types. Practically speaking, we can read this as saying that our experiment has *10* events, occurring at times $t_1, \ldots, t_{10}$ with event types $a_1, \ldots, a_{10}$.

Typically, the events occur in groups, say odd events are labelled a, even ones b. We might rewrite this as

$$E = \delta_{(t_1,a)} + \delta_{(t_2,b)} + \delta_{(t_3,a)} + \cdots + \delta_{t_{10},b} \tag{3.2}$$

This type of experiment can be represented by two counting processes (E_a, E_b) defined as

$$\begin{aligned} E_a(t) &= \sum_{t_j, j \text{ odd}} 1_{\{t_j \leq t\}} \\ E_b(t) &= \sum_{t_j, j \text{ even}} 1_{\{t_j \leq t\}} \end{aligned} \tag{3.3}$$

These delta-function responses are effectively events of duration 0 and infinite height.

For block designs, we might also allow event durations, which fit nicely into the counting processes $(E_a(t), E_b(t))$.

Suppose that the presentations above each had a duration of 20 seconds, the counting processes could look like

$$\begin{aligned} E_a(t) &= \sum_{t_j, j \text{ odd}} \frac{1}{20} \int_{t_j}^{\min(t_j+20, t)} ds \\ E_b(t) &= \sum_{t_j, j \text{ even}} \frac{1}{20} \int_{t_j}^{\min(t_j+20, t)} ds \end{aligned} \tag{3.4}$$

Counting processes vs. intensities

Though the experiment can be represented in terms of the pair $(E_a(t), E_b(t))$, it is a little easier, and more common in neuroimaging applications to work with the derivatives

$$e_a(t) = \frac{\partial}{\partial t} E_a(t), \qquad e_b(t) = \frac{\partial}{\partial t} E_b(t) \tag{3.5}$$

Continuous stimuli

Some experiments do not fit well into this "event-type" paradigm but are, rather, more continuous in nature. For instance, a rotating checkerboard, for which orientation, contrast, are functions of experiment time *t*. This experiment can be represented in terms of a state vector $(O(t), C(t))$.

3.3.2 Neuronal model

The neuronal model is a model of the activity as a function of *t* at a neuron *x* given the experimental model E. For instance, one model could be

$$N^1_{x,t} = \beta_{a,x} e_a(t) + \beta_{b,x} e_b(t) \tag{3.6}$$

This states that the neuronal response is a delta function, with identical heights for each trial type of *a*, and *b*, respectively. An alternative model is for the height of each event to decay exponentially immediately after each stimulus presentation. Mathematically, this can be represented by convolution with an exponential kernel as

$$N^2_{x,t} = \beta_{a,x} \int_0^\infty e_a(t-s) e^{-\theta_a s}\, ds + \beta_{b,x} \int_0^\infty e_b(t-s) e^{-\theta_b s}\, ds \tag{3.7}$$

Another model, perhaps less plausible scientifically, might keep delta functions, but have the height of the spikes be a function of experiment time, perhaps decreasing exponentially.

$$N^3_{x,t} = \beta_{a,x} \frac{\partial}{\partial t} \int_{-\infty}^t e^{-\theta_a s} dE_a(s) + \beta_{b,x} \frac{\partial}{\partial t} \int_{-\infty}^t e^{-\theta_b s} dE_b(s) \tag{3.8}$$

This model states that the neuronal activity decreases exponentially in time within each event type, with a time scale specific to each group.

Note that each of these neuronal models are linear operators of the pair (E_a, E_b) though some have nonlinear parameters, i.e. the timeconstants (θ_a, θ_b). The inputs are timecourses, and the output is a timecourse representing the neuronal activity at neuron *x* as a function of experiment time *t*.

Continuous stimuli

In our continuous example above, a reasonable neuronal model might be

$$N^1_{x,t} = \beta_{O,x} O(t) + \beta_{C,x} C(t) \tag{3.9}$$

Allowing for possible time shifts for both orientation and contrast, another model might be

$$N^2_{x,t} = \beta_{O,x} O(t - \tau_O) + \beta_{C,x} C(t - \tau_C) \tag{3.10}$$

Note that this model is linear in the pair $(O(t), C(t))$, but has two nonlinear parameters (τ_O, τ_C).

A third model, could incorporate derivative information of $(O(t), C(t))$

$$N^3_{x,t} = \beta_{O,0,x} O(t) + \beta_{O,1,x} \dot{O}(t) + \beta_{C,0,x} C(t) + \beta_{C,1,x} \dot{C}(t) \tag{3.11}$$

where $\dot{f}(t) = \partial f / \partial t$.

CHAPTER

FOUR

GLOSSARY

AFNI AFNI is a functional imaging analysis package. It is funded by the NIMH, based in Bethesda, Maryland, and directed by Robert Cox. Like *FSL*, it is written in C, and it's very common to use shell scripting of AFNI command line utilities to automate analyses. Users often describe liking AFNI's scriptability, and image visualization. It uses the *GPL* license.

BSD Berkeley software distribution license. The BSD license is permissive, in that it allows you to modify and use the code without requiring that you use the same license. It allows you to distribute closed-source binaries.

BOLD Contrast that is blood oxygen level dependent. When a brain area becomes active, blood flow increases to that area. It turns out that, with the blood flow increase, there is a change in the relative concentrations of oxygenated and deoxygenated hemoglobin. Oxy- and deoxy- hemoglobin have different magnetic properties. This in turn leads to a change in MRI signal that can be detected by collecting suitably sensitive MRI images at regular short intervals during the blood flow chance. See the the wikipedia FMRI article for more detail.

BrainVisa BrainVISA is a sister project to NIPY. It also uses Python, and provides a carefully designed framework and automatic GUI for defining imaging processing workflows. It has tools to integrate command line and other utilities into these workflows. Its particular strength is anatomical image processing but it also supports FMRI and other imaging modalities. BrainVISA is based in NeuroSpin, outside Paris.

DTI Diffusion tensor imaging. DTI is rather poorly named, because it is a model of the diffusion signal, and an analysis method, rather than an imaging method. The simplest and most common diffusion tensor model assumes that diffusion direction and velocity at every voxel can be modeled by a single tensor - that is, by an ellipse of regular shape, fully described by the length and orientation of its three orthogonal axes. This model can easily fail in fairly common situations, such as white-matter fiber track crossings.

DWI Diffusion-weighted imaging. DWI is the general term for MRI imaging designed to image diffusion processes. Sometimes reseachers use *DTI* to have the same meaning, but *DTI* is a common DWI signal model and analysis method.

EEGlab The most widely-used open-source package for analyzing electrophysiological data. EEGlab is written in *matlab* and uses a *GPL* license.

FMRI Functional magnetic resonance imaging! It refers to MRI image acquisitions and analysis designed to look at brain function rather than structure. Most people use FMRI to refer to *BOLD* imaging in particular. See the wikipedia FMRI article for more detail.

FSL FSL is the FMRIB software library, written by the FMRIB analysis group, and directed by Steve Smith. Like *AFNI*, it is a large collection of C / C++ command line utilities that can be scripted with a custom GUI / batch system, or using shell scripting. Its particular strength is analysis of *DWI* data, and *ICA* functional data analysis, although it has strong tools for the standard *SPM approach* to FMRI. It is free for academic use, and open-source, but not free for commercial use.

GPL The GPL is the GNU general public license. It is one of the most commonly-used open-source sofware licenses. The distinctive feature of the GPL license is that it requires that any code derived from GPL code also uses a GPL

license. It also requires that any code that is statically or dynamically linked to GPL code has a GPL-compatible license. See: http://en.wikipedia.org/wiki/GNU_General_Public_License and http://www.gnu.org/licenses/gpl-faq.html

ICA Independent component analysis is a multivariate technique related to *PCA*, to estimate independent components of signal from multiple sensors. In functional imaging, this usually means detecting underlying spatial and temporal components within the brain, where the brain voxels can be considered to be different sensors of the signal. See the wikipedia ICA page.

LGPL The lesser GNU public license. LGPL differs from the *GPL* in that you can link to LGPL code from non-LGPL code without having to adopt a GPL-compatible license. However, if you modify the code (create a "derivative work"), that modification has to be released under the LGPL. See wikipedia LGPL for more discussion.

Matlab matlab began as a high-level programming language for working with matrices. Over time it has expanded to become a fairly general-purpose language. See also: http://en.wikipedia.org/wiki/MATLAB. It has good numerical algorithms, 2D graphics, and documentation. There are several large neuroscience software projects wtitten in matlab, including *SPM software*, and *EEGlab*.

PCA Principal component analysis is a multivariate technique to determine orthogonal components across multiple sources (or sensors). See *ICA* and the wikipedia PCA page.

PET Positron emission tomography is a nethod of detecting the spatial distributions of certain radiolabeled compounds - usually in the brain. The scanner detectors pick up the spatial distribution of emitted radiation from within the body. From this pattern, it is possible to reconstruct the distribution of radiactivity in the body, using techniques such as filtered back projection. PET was the first mainstream technique used for detecting regional changes in blood-flow as an index of which brain areas were active when the subject is doing various tasks, or at rest. These studies nearly all used *water activation PET*. See the wikipedia PET entry.

SPM SPM (statistical parametric mapping) refers either to the *SPM approach* to analysis or the *SPM software* package.

SPM approach Statistical parametric mapping is a way of analyzing data, that involves creating an image (the *map*) containing statistics, and then doing tests on this statistic image. For example, we often create a t statistic image where each *voxel* contains a t statistic value for the time-series from that voxel. The *SPM software* package implements this approach - as do several others, including *FSL* and *AFNI*.

SPM software SPM (statistical parametric mapping) is the name of the matlab based package written by John Ashburner, Karl Friston and others at the Functional Imaging Laboratory in London. More people use the SPM package to analyze *FMRI* and *PET* data than any other. It has good lab and community support, and the *matlab* source code is available under the *GPL* license.

VoxBo Quoting from the Voxbo webpage - "VoxBo is a software package for the processing, analysis, and display of data from functional neuroimaging experiments". Like *SPM*, *FSL* and *AFNI*, VoxBo provides algorithms for a full FMRI analysis, including statistics. It also provides software for lesion-symptom analysis, and has a parallel scripting engine. VoxBo has a *GPL* license. Dan Kimberg leads development.

voxel Voxels are volumetric pixels - that is, they are values in a regular grid in three dimensional space - see http://en.wikipedia.org/wiki/Voxel

water activation PET A *PET* technique to detect regional changes in blood flow. Before each scan, we inject the subject with radiolabeled water. The radiolabeled water reaches the arterial blood, and then distributes (to some extent) in the brain. The concentration of radioactive water increases in brain areas with higher blood flow. Thus, the image of estimated counts in the brain has an intensity that is influenced by blood flow. This use has been almost completely replaced by the less invasive *BOLD FMRI* technique.

Part II

Developer Guide

CHAPTER

FIVE

DEVELOPMENT QUICKSTART

5.1 Source Code

NIPY uses launchpad for our code hosting. For immediate access to the source code, see the nipy launchpad site.

5.2 Guidelines

We have adopted many developer guidelines in an effort to make development easy, and the source code readable, consistent and robust. Many of our guidelines are adopted from the scipy / numpy community. We welcome new developers to the effort, if you're interested in developing code or documentation please join the nipy mailing list and introduce yourself. If you plan to do any code development, we ask that you take a look at the following guidelines. We do our best to follow these guidelines ourselves:

- *How to write documentation* : Documentation is critical. This document describes the documentation style, syntax, and tools we use.
- Numpy/Scipy Coding Style Guidelines: This is the coding style we strive to maintain.
- *nipy bazaar workflow* : This describes our process for version control.
- *Testing* : We've adopted a rigorous testing framework.
- *Optimization*: "premature optimization is the root of all evil."

5.3 Checking out the latest version

To check out the latest version of nipy you need bzr version greater than 0.92:

```
bzr branch lp:nipy
```

There are two methods to install a development version of nipy. For both methods, build the extensions in place:

```
python setup.py build_ext --inplace
```

Then you can either:

1. Use the `mynipy` script in the tools directory of the nipy source. There are directions and examples in the docstring of that file, but basically it updates a symbolic link in your *site-packages* directory to the inplace build of your source. The advantage of this method is it does not require any modifications of your PYTHONPATH.

2. Place the source directory in your PYTHONPATH.

With either method, all of the modifications made to your source tree will be picked up when nipy is imported.

5.4 Submitting a patch

The preferred method to submit a patch is to create a branch of nipy on your machine, modify the code and push that branch to your launchpad directory. Then email the list and we will review your code and hopefully apply (merge) your patch. See the instructions for *Initializing your development environment*.

If you do not wish to use bazaar and launchpad, please feel free to file a bug report and submit a patch or email the nipy mailing list.

5.5 Bug reports

If you find a bug in nipy, please submit a bug report at the nipy bugs launchpad site so that we can fix it.

CHAPTER

SIX

DEVELOPER INSTALLS FOR DIFFERENT DISTRIBUTIONS

6.1 Ubuntu / debian developer install

See *Download and Install*

This assumes a recent (Ubuntu >= 8.04) version.

Requirements:

```
sudo apt-get build-essential
sudo apt-get install python-dev
sudo apt-get install python-numpy python-numpy-dev python-scipy
```

Options:

```
sudo apt-get install ipython
sudo apt-get install python-matplotlib
```

For getting the code via version control:

```
sudo apt-get install bzr
```

Then follow the instructions at *Checking out the latest version*

6.2 Fedora developer install

See *Download and Install*

This assumes a recent Fedora (>=10) version. It may work for earlier versions - see *Download and Install* for requirements.

This page may also hold for Fedora-based distributions such as Mandriva and Centos.

Run all the `yum install` commands as root.

Requirements:

```
yum install gcc-c++
yum install python-devel
yum install numpy scipy
```

Options:

```
yum install ipython
yum install python-matplotlib
```

For getting the code via version control:

```
yum install bzr
```

Then follow the instructions at *Checking out the latest version*

6.3 Development install on windows

The easiest way to get the dependencies is to install the Enthought Tool Suite . This gives you MinGW, Python, Numpy, Scipy, ipython and matplotlib. If instead you want to do it by component, try the instructions below.

Requirements:

- Download and install MinGW
- Download and install the windows binary for Python
- Download and install the Numpy and Scipy binaries

Options:

- Download and install ipython, being careful to follow the windows installation instructions
- Download and install matplotlib

Alternatively, if you are very brave, you may want to install numpy / scipy from source - see our maybe out of date *Building Scipy/Numpy on Windows with Optimized Numerical Libraries* for details.

Whether you used ETS or the instructions above, you will next need to get the NIPY code via version control:

- Download and install the windows binary for bazaar
- Go to the windows menu, find the *bazaar* menu, and run `bzr` in a windows terminal.

You should now be able to follow the instructions in *Checking out the latest version*, but with the following modifications:

First, for the `python setup.py` steps, you will need to add the `--compiler=mingw32` flag, like this:

```
python setup.py build_ext --inplace --compiler=mingw32
```

To build a windows installer, use:

```
python setup.py bdist_wininst
```

Second, you may want to use the ipython shell to work in. For system commands use the `!` escape, like this, from the ipython prompt:

```
!python setup.py build_ext --inplace --compiler=mingw32
```

CHAPTER

SEVEN

DEVELOPMENT GUIDELINES

7.1 How to write documentation

Nipy uses the Sphinx documentation generating tool. Sphinx translates reST formatted documents into html and pdf documents. All our documents and docstrings are in reST format, this allows us to have both human-readable docstrings when viewed in ipython, and web and print quality documentation.

7.1.1 Building the documentation

You need to have Sphinx (version 0.5 or greater), graphviz (version 2.20 or greater) and the svn of matplotlib (0.98.6svn) installed in order to build the documentation. (*Note: The svn requirement of matplotlib will go away after the next release of matplotlib.*)

The `Makefile` (in the top-level doc directory) automates the generation of the documents. To make the HTML documents:

```
make html
```

For PDF documentation do:

```
make pdf
```

The built documentation is then placed in a `build/html` or `build/latex` subdirectories.

For more options, type:

```
make help
```

7.1.2 Viewing the documentation

We also build our website using sphinx. All of the documentation in the `docs` directory is included on the website. There are a few files that are website only and these are placed in the `www` directory. The easiest way to view the documentation while editing is to build the website and open the local build in your browser:

```
make web
```

Then open `www/build/html/index.html` in your browser.

7.1.3 Syntax

Please have a look at our *Sphinx Cheat Sheet* for examples on using Sphinx and reST in our documentation.

The Sphinx website also has an excellent sphinx rest primer.

Additional reST references::

- reST primer
- reST quick reference

Consider using emacs for editing rst files - see *ReST mode*

7.1.4 Style

Nipy has adopted the numpy documentation standards. The numpy coding style guideline is the main reference for how to format the documentation in your code. It's also useful to look at the source reST file that generates the coding style guideline.

Numpy has a detailed example for writing docstrings.

7.1.5 Documentation Problems

See our *Documentation FAQ* if you are having problems building or writing the documentation.

7.2 Sphinx Cheat Sheet

Wherein I show by example how to do some things in Sphinx (you can see a literal version of this file below in *This file*)

7.2.1 Making a list

It is easy to make lists in rest

Bullet points

This is a subsection making bullet points

- point A
- point B
- point C

Enumerated points

This is a subsection making numbered points

1. point A
2. point B
3. point C

7.2.2 Making a table

This shows you how to make a table – if you only want to make a list see *Making a list*.

Name	Age
John D Hunter	40
Cast of Thousands	41
And Still More	42

7.2.3 Making links

Cross-references sections and documents

Use reST labels to cross-reference sections and other documents. The mechanism for referencing another reST document or a subsection in any document, including within a document are identical. Place a *reference label* above the section heading, like this:

```
.. _sphinx_helpers:

====================
 Sphinx Cheat Sheet
====================
```

Note the blank line between the *reference label* and the section heading is important!

Then refer to the *reference label* in another document like this:

```
:ref:`sphinx_helpers`
```

The reference is replaced with the section title when Sphinx builds the document while maintaining the linking mechanism. For example, the above reference will appear as *Sphinx Cheat Sheet*. As the documentation grows there are many references to keep track of.

For documents, please use a *reference label* that matches the file name. For sections, please try and make the *refence label* something meaningful and try to keep abbreviations limited. Along these lines, we are using *underscores* for multiple-word *reference labels* instead of hyphens.

Sphinx documentation on Cross-referencing arbitrary locations has more details.

External links

For external links you are likely to use only once, simple include the like in the text. This link to google was made like this:

```
`google <http://www.google.com>`_
```

For external links you will reference frequently, we have created a `links_names.txt` file. These links can then be used throughout the documentation. Links in the `links_names.txt` file are created using the reST reference syntax:

```
.. _targetname: http://www.external_website.org
```

To refer to the reference in a separate reST file, include the `links_names.txt` file and refer to the link through it's target name. For example, put this include at the bottom of your reST document:

```
.. include:: ../links_names.txt
```

and refer to the hyperlink target:

```
blah blah blah targetname_ more blah
```

Links to classes, modules and functions

You can also reference classes, modules, functions, etc that are documented using the sphinx autodoc facilites. For example, see the module `matplotlib.backend_bases` documentation, or the class `LocationEvent`, or the method `mpl_connect()`.

7.2.4 ipython sessions

Michael Droettboom contributed a sphinx extension which does pygments syntax highlighting on ipython sessions

```
In [69]: lines = plot([1,2,3])

In [70]: setp(lines)
  alpha: float
  animated: [True | False]
  antialiased or aa: [True | False]
  ...snip
```

This support is included in this template, but will also be included in a future version of Pygments by default.

7.2.5 Formatting text

You use inline markup to make text *italics*, **bold**, or `monotype`.

You can represent code blocks fairly easily:

```
import numpy as np
x = np.random.rand(12)
```

Or literally include code:

```
import matplotlib.pyplot as plt
plt.plot([1,2,3], [4,5,6])
plt.ylabel('some more numbers')
```

7.2.6 Using math

In sphinx you can include inline math $x \leftarrow y\, x \forall y\, x - y$ or display math

$$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \frac{1}{8\pi 2} \int_{\alpha_2}^{\alpha_2} d\alpha_2' \left[\frac{U^{2\beta}_{\delta_1 \rho_1} - \alpha_2' U^{1\beta}_{\rho_1 \sigma_2}}{U^{0\beta}_{\rho_1 \sigma_2}} \right] \tag{7.1}$$

This documentation framework includes a Sphinx extension, `sphinxext/mathmpl.py`, that uses matplotlib to render math equations when generating HTML, and LaTeX itself when generating a PDF. This can be useful on systems that have matplotlib, but not LaTeX, installed. To use it, add `mathpng` to the list of extensions in `conf.py`.

Current SVN versions of Sphinx now include built-in support for math. There are two flavors:

- pngmath: uses dvipng to render the equation
- jsmath: renders the math in the browser using Javascript

To use these extensions instead, add `sphinx.ext.pngmath` or `sphinx.ext.jsmath` to the list of extensions in `conf.py`.

All three of these options for math are designed to behave in the same way.

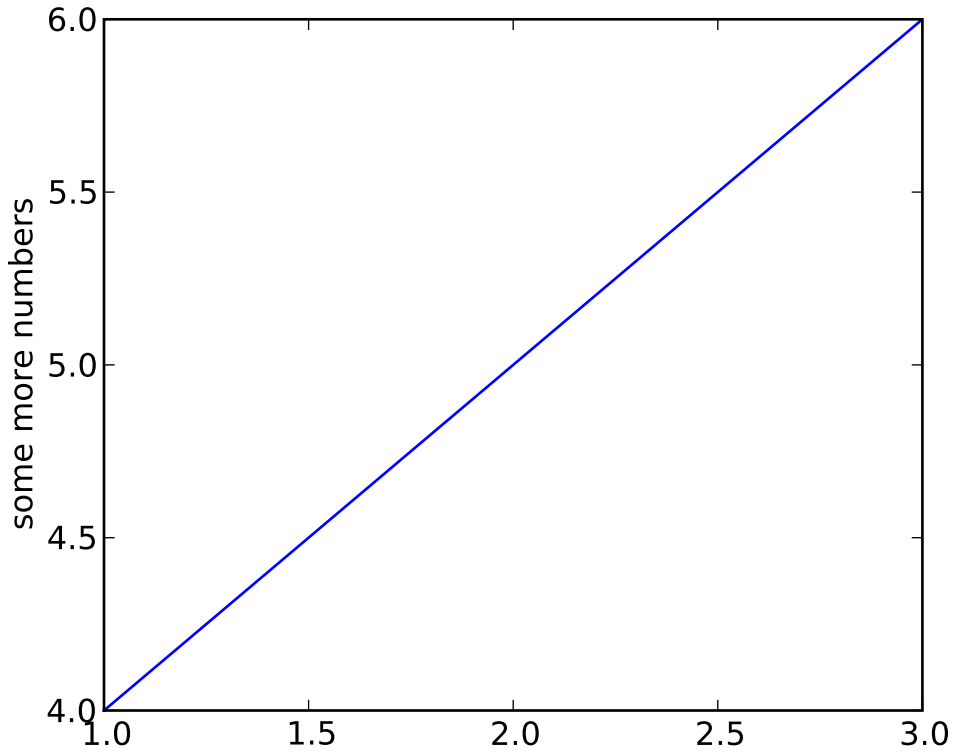
7.2.7 Inserting matplotlib plots

Inserting automatically-generated plots is easy. Simply put the script to generate the plot in any directory you want, and refer to it using the `plot` directive. All paths are considered relative to the top-level of the documentation tree. To include the source code for the plot in the document, pass the `include-source` parameter:

```
.. plot:: devel/guidelines/elegant.py
   :include-source:
```

In the HTML version of the document, the plot includes links to the original source code, a high-resolution PNG and a PDF. In the PDF version of the document, the plot is included as a scalable PDF.

```
import matplotlib.pyplot as plt
plt.plot([1,2,3], [4,5,6])
plt.ylabel('some more numbers')
```



7.2.8 Emacs helpers

See *ReST mode*

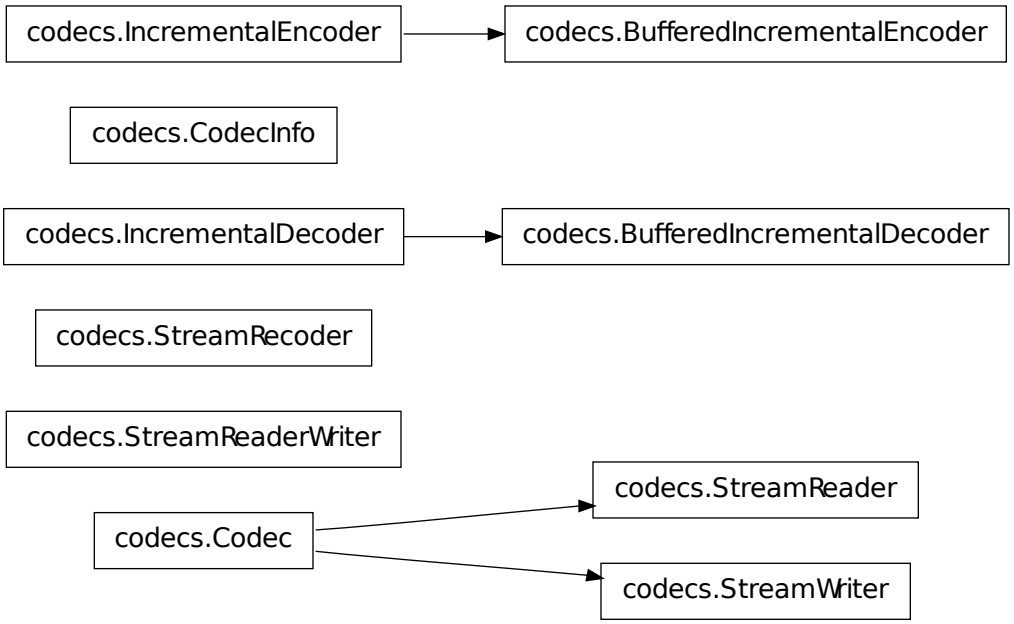
7.2.9 Inheritance diagrams

Inheritance diagrams can be inserted directly into the document by providing a list of class or module names to the `inheritance-diagram` directive.

For example:

```
.. inheritance-diagram:: codecs
```

produces:



7.2.10 This file

```
.. _sphinx_helpers:

====================
 Sphinx Cheat Sheet
====================

Wherein I show by example how to do some things in Sphinx (you can see
a literal version of this file below in :ref:`sphinx_literal`)
```

```
.. _making_a_list:

Making a list
-------------

It is easy to make lists in rest

Bullet points
^^^^^^^^^^^^^

This is a subsection making bullet points

* point A

* point B

* point C


Enumerated points
^^^^^^^^^^^^^^^^^

This is a subsection making numbered points

#. point A

#. point B

#. point C


.. _making_a_table:

Making a table
--------------

This shows you how to make a table -- if you only want to make a list
see :ref:`making_a_list`.

==================   ============
Name                 Age
==================   ============
John D Hunter        40
Cast of Thousands    41
And Still More       42
==================   ============

.. _making_links:

Making links
------------

Cross-references sections and documents
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Use reST labels to cross-reference sections and other documents. The
mechanism for referencing another reST document or a subsection in any
document, including within a document are identical. Place a
```

```
*reference label* above the section heading, like this::

        .. _sphinx_helpers:

        ====================
         Sphinx Cheat Sheet
        ====================

Note the blank line between the *reference label* and the section
heading is important!

Then refer to the *reference label* in another
document like this::

     :ref:`sphinx_helpers`

The reference is replaced with the section title when Sphinx builds
the document while maintaining the linking mechanism.  For example,
the above reference will appear as :ref:`sphinx_helpers`.  As the
documentation grows there are many references to keep track of.

For documents, please use a *reference label* that matches the file
name.  For sections, please try and make the *refence label* something
meaningful and try to keep abbreviations limited.  Along these lines,
we are using *underscores* for multiple-word *reference labels*
instead of hyphens.

Sphinx documentation on `Cross-referencing arbitrary locations
<http://sphinx.pocoo.org/markup/inline.html#cross-referencing-arbitrary-locations>`_
has more details.

External links
^^^^^^^^^^^^^^

For external links you are likely to use only once, simple include the
like in the text.  This link to `google <http://www.google.com>`_ was
made like this::

     `google <http://www.google.com>`_

For external links you will reference frequently, we have created a
``links_names.txt`` file.  These links can then be used throughout the
documentation.  Links in the ``links_names.txt`` file are created
using the `reST reference
<http://docutils.sourceforge.net/docs/user/rst/quickref.html#hyperlink-targets>`_
syntax::

        .. _targetname: http://www.external_website.org

To refer to the reference in a separate reST file, include the
``links_names.txt`` file and refer to the link through it's target
name.  For example, put this include at the bottom of your reST
document::

     .. include:: ../links_names.txt

and refer to the hyperlink target::
```

```
    blah blah blah targetname_ more blah



Links to classes, modules and functions
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

You can also reference classes, modules, functions, etc that are
documented using the sphinx `autodoc
<http://sphinx.pocoo.org/ext/autodoc.html>`_ facilites.  For example,
see the module :mod:`matplotlib.backend_bases` documentation, or the
class :class:`~matplotlib.backend_bases.LocationEvent`, or the method
:meth:`~matplotlib.backend_bases.FigureCanvasBase.mpl_connect`.

.. _ipython_highlighting:

ipython sessions
----------------

Michael Droettboom contributed a sphinx extension which does pygments
syntax highlighting on ipython sessions

.. sourcecode:: ipython

    In [69]: lines = plot([1,2,3])

    In [70]: setp(lines)
      alpha: float
      animated: [True | False]
      antialiased or aa: [True | False]
      ...snip

This support is included in this template, but will also be included
in a future version of Pygments by default.

.. _formatting_text:

Formatting text
---------------

You use inline markup to make text *italics*, **bold**, or ``monotype``.

You can represent code blocks fairly easily::

   import numpy as np
   x = np.random.rand(12)

Or literally include code:

.. literalinclude:: elegant.py


.. _using_math:

Using math
----------

In sphinx you can include inline math :math:`x\leftarrow y\ x\forall
```

```
y\ x-y` or display math

.. math::

  W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \frac{1}{8 \pi 2} \int^{\alp

This documentation framework includes a Sphinx extension,
:file:`sphinxext/mathmpl.py`, that uses matplotlib to render math
equations when generating HTML, and LaTeX itself when generating a
PDF.  This can be useful on systems that have matplotlib, but not
LaTeX, installed.  To use it, add ``mathpng`` to the list of
extensions in :file:`conf.py`.

Current SVN versions of Sphinx now include built-in support for math.
There are two flavors:

  - pngmath: uses dvipng to render the equation

  - jsmath: renders the math in the browser using Javascript

To use these extensions instead, add ``sphinx.ext.pngmath`` or
``sphinx.ext.jsmath`` to the list of extensions in :file:`conf.py`.

All three of these options for math are designed to behave in the same
way.

Inserting matplotlib plots
--------------------------

Inserting automatically-generated plots is easy.  Simply put the script to
generate the plot in any directory you want, and refer to it using the ``plot``
directive.  All paths are considered relative to the top-level of the
documentation tree.  To include the source code for the plot in the document,
pass the ``include-source`` parameter::

  .. plot:: devel/guidelines/elegant.py
     :include-source:

In the HTML version of the document, the plot includes links to the
original source code, a high-resolution PNG and a PDF.  In the PDF
version of the document, the plot is included as a scalable PDF.

.. plot:: devel/guidelines/elegant.py
   :include-source:

Emacs helpers
-------------

See :ref:`rst_emacs`

Inheritance diagrams
--------------------

Inheritance diagrams can be inserted directly into the document by
providing a list of class or module names to the
``inheritance-diagram`` directive.

For example::
```

```
  .. inheritance-diagram:: codecs

produces:

.. inheritance-diagram:: codecs

.. _sphinx_literal:

This file
---------

.. literalinclude:: sphinx_helpers.rst
```

7.3 nipy bazaar workflow

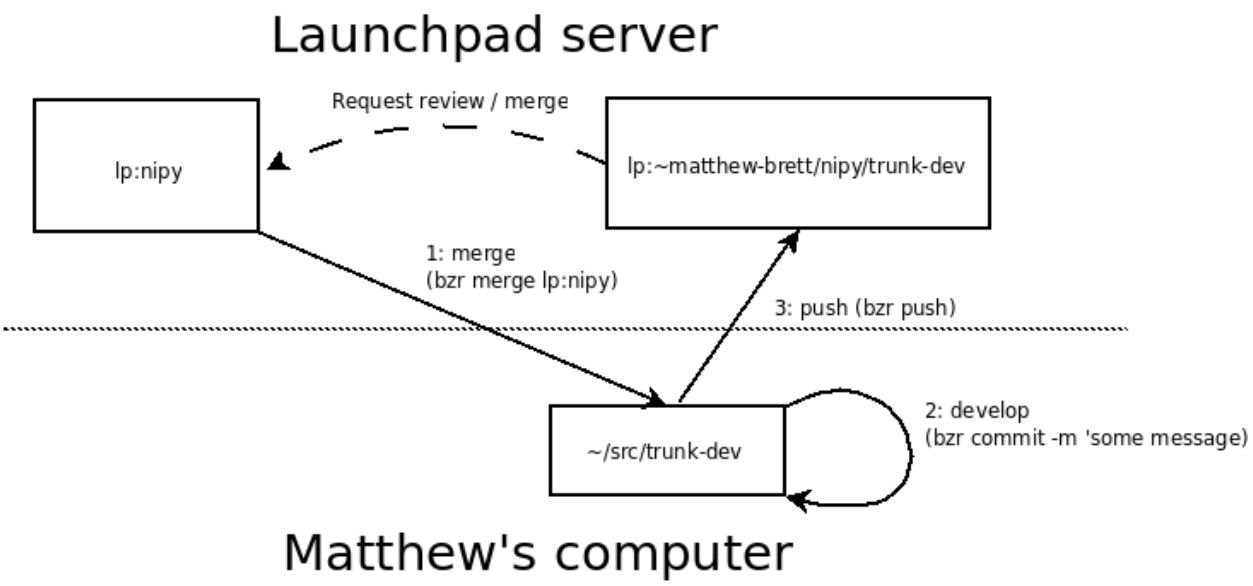
Contents

7.3.1 Overview

This document describes the standard workflow for nipy development using bazaar and the launchpad hosting service. It would help very much in keeping track of shared code if all nipy developers adopt this workflow. The workflow should be flexible enough that even an occasional developer can contribute code to nipy since it does not require all contributors to have commit access to the main trunk. This particular workflow was developed to achieve these primary objectives.

1. Maintain a linear revision number history on the main nipy-trunk.
2. Provide a clear process for code reviews.
3. Minimize damage to the trunk during large code transitions.

The diagram describes the usual workflow:



We describe the various parts of this diagram in the following sections.

7.3.2 Launchpad structure

On launchpad, there exists one main nipy trunk that only an administrator has commit access to (see *nipy bazaar administration*). All developers have their own launchpad.net/~developer directory which may be registered with the nipy project. In addition, there may be shared team directories on nipy, for feature development, but in general individual development should happen in your ~developer directory.

A snapshot of the nipy code directories on launchpad

- lp:nipy - the nipy-trunk
- ~cburns/nipy - developer branch for cburns (registered with nipy)
- ~twaite/nipy - developer branch for twaite (registered with nipy)

7.3.3 Developer directory structure

Your development directory will contain at least one branch:

- nipy-repo/trunk-dev

where *trunk-dev* is a branch from your user directory on launchpad and is your working development branch.

7.3.4 Initializing your development environment

We're using the shared repository feature of Bazaar and a Decentralized with human gatekeeper workflows. The basic process is as follows:

1. Create an account on launchpad, if you don't have one already. Follow the *Log in / Register* link on the top of the page.
2. Create a shared repository:

```
bzr init-repo --trees nipy-repo
cd nipy-repo
```

3. **Tell bazaar who you are::** bzr whoami "Barack Obama <bobama@whitehouse.gov>"
4. Create your own personal development branch, named *trunk-dev*, by replicating the nipy trunk:

   ```
   bzr branch lp:nipy trunk-dev
   ```

5. Push your development branch up to launchpad for backup and to make your changes visible to the rest of the team:

   ```
   cd trunk-dev
   bzr push bzr+ssh://USER@bazaar.launchpad.net/~USER/nipy/trunk-dev --remember
   ```

   Real example of the line above for user *cburns*:

   ```
   bzr push bzr+ssh://cburns@bazaar.launchpad.net/~cburns/nipy/trunk-dev --remember
   ```

You now have your own branch, stored on launchpad, and on your machine, in which you can develop nipy code. If you get an error while trying to push, see the *Push Error: Permission Denied (publickey)* document for instructions on launchpad and your ssh key.

7.3.5 Daily development cycle

You will usually want to keep track of changes in the nipy trunk. To do this, you will often want to merge the nipy trunk into your development branch:

```
cd trunk-dev
bzr merge lp:nipy
bzr commit -m 'Merge from trunk'
```

When editing you will want to push your changes up to your launchpad branch:

```
cd trunk-dev
# edit code
bzr ci -m "Add a meaningful summary of your edits in one complete sentence."
# repeat
...
# periodically push to lp:~user (like end of day)
bzr push
```

7.3.6 Keeping current with the nipy trunk

Since we are all doing development in our own launchpad branches, each developer should keep their branch up-to-date with the main trunk so their branch doesn't diverge significantly from the rest of the team. This is done with the `bzr merge lp:nipy` line in the *Daily development cycle* section above. Launchpad allows developers to subscribe to a branch and receive email notifications when there are changes. By subscribing to the nipy trunk, you will be emailed whenever the trunk changes, reminding you to merge the trunk into your branch.

On the nipy trunk page there is an **Owner / Subscribers** section on the right-hand side. Click on the link *Subscribe yourself*. Select the desired options and click the *Subscribe* button.

These options should be sufficient for developers:

Notification Level: Branch attribute and revision notifications

Generated Diff Size Limit: Don't send diffs (*more below*)

Code review Level: Email about all changes

The *Diff Size Limit* is a personal preference. The full diff's can be long and a lot to read in an email. I find it easier to view the changesets on Launchpad with the color-coded diffs.

Note: Developers can also subscribe to other team members branches.

7.3.7 Code review of developer branches

In order to do code reviews, each developer should propose their developer branch for merging into the nipy mainline. We have decided to keep these proposed merges open indefinitely so the code reviewer can use Launchpad to view branch changes prior to pulling the branch. *You only need to do this procedure once.*

In your developer code directory, select the **Propose for merging into another branch** link and choose these options:

Target Branch: The NIPY mainline (*default selection*)

Click on the **Register** button.

Edit the branch **Title** and **Summary**. In your branch page, select the little yellow circle with the pencil icon next to the title to *Change branch details*. Set the options to something like:

Title: Chris' development copy of the nipy trunk

Summary: This branch is my development copy of the trunk. It is available here for review by other developers, and merges will be periodically made into trunk.

For this reason, it will always be marked for review to be merged.

Status: Development

Click on the **Change branch** button to finalize.

Look at Chris' branch for an example.

Bazaar Documentation

The bzr documentation is thorough and excellent.

Benjamin Thyreau also kindly posted a bzr tutorial in French.

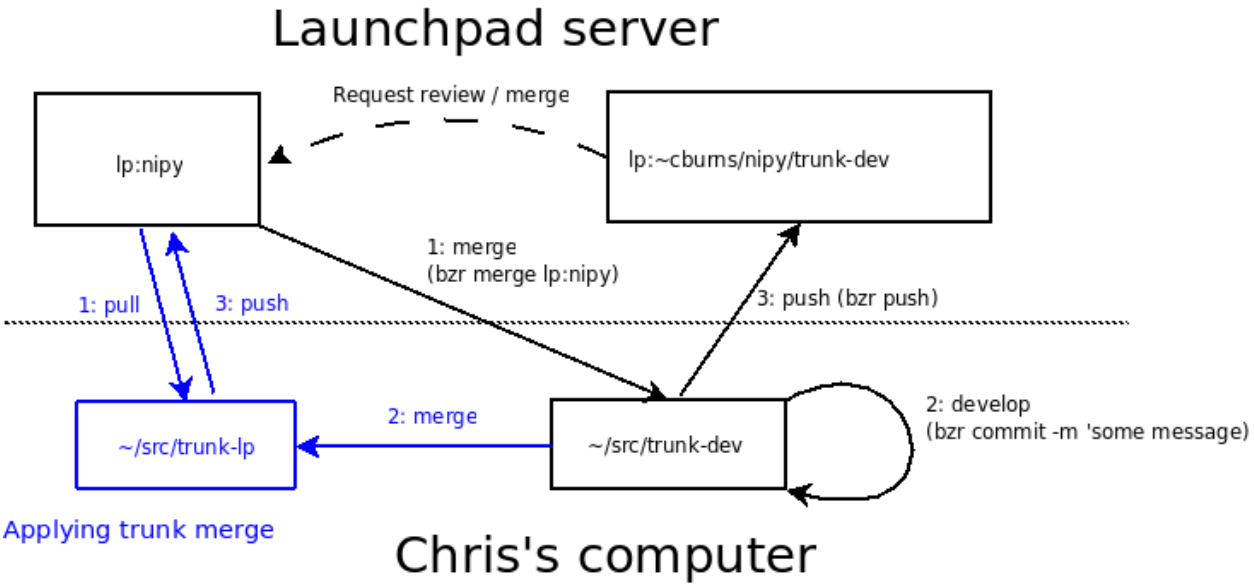
And we've start a collection of *Bazaar Tips* for some common questions.

7.4 nipy bazaar administration

Contents

7.4.1 Submit reviewed code

The administrator (at the moment, this is Chris Burns), will merge changes into the trunk, from development branches. The diagram shows the *administrator* workflow:



The process looks something like this on the command line:

```
cd trunk-lp
bzr pull lp:nipy
bzr merge ../trunk-dev
bzr ci --author="user name"
# enter detailed commit message in a docstring format
bzr push lp:nipy
```

Please don't do this yourself; even the core members of the development team will leave this process to the administrator.

7.4.2 Example: Merging Fernando's Matplotlib Sphinx docs

One of the first test cases of the trunk-lp/trunk-dev branch strategy was merging the matplotlib documentation skeleton, which uses sphinx into the nipy-trunk. The summary of the process:

1. Pull the nipy trunk and confirm your main trunk is current.
2. Download Fernando's branch.
3. Build and test Fernando's branch.
4. Merge Fernando's trunk into the main trunk.
5. Build and test main trunk after the merge.
6. Commit merge and push to launchpad

Matthew Brett performed this merge on his machine.

```
# Change to nipy source repository
cd ~/dev_trees/nipy-repo

# Update main trunk
cd lp-trunk/
bzr pull

# Download Fernando branch
bzr branch lp:~fperez/nipy/trunk-dev fp-trunk-dev
cd ..

# Build and test.  # FIXME: Need a good system for doing this
#python setup.py build
#python setup.py install
#python -c "import nipy as ni; ni.test()"

# Merge Fernando branch
bzr log -r last:
bzr merge ../fp-trunk
bzr commit

# Push up to launchpad
bzr push bzr+ssh://matthew-brett@bazaar.launchpad.net/~nipy-developers/nipy/trunk --remember
```

7.5 Bazaar Tips

There are a few adjustments in switching from a svn workflow to a bzr workflow. This page is meant to catalog some of those gotcha's and help all of us transition.

7.5.1 Nipy's current workflow

Most of us are using a distributed workflow process in what bazaar calls the *Decentralized with human gatekeeper*. The bazaar documenation on workflows explains this process and the various workflows possible with bazaar.

The Bazaar User Guide also has more details.

7.5.2 Pushing code to launchpad

When you want to push your changes you must specify the shared mainline to push to. The `remember` flag will make bzr store this location so later you only need `$ bzr push`:

```
$ bzr push bzr+ssh://USERNAME@bazaar.launchpad.net/~USERNAME/nipy/trunk-dev --remember
```

7.5.3 Push Error: Permission Denied (publickey)

If you get this error when you try to push your changes you need to upload your ssh key to Launchpad. Bazarr has a SSH page explaining some of the details.

Basically you create your public key with `ssh-keygen`, then login to your launchpad site, click on the *Change details* link and follow the *Update SSH keys* link in the menu and upload your key.

7.5.4 What changes am I about to submit?

You've been making changes over several days and want to see what you've done before merging with the mainline:

```
cburns@nipy 12:44:46 $ bzr diff -r submit:
Using parent branch http://bazaar.launchpad.net/%7Enipy-developers/nipy/trunk/
```

7.5.5 What was my last commit?

This will print out the log for the last commit. A meaningful commit message helps here:

```
cburns@nipy 12:44:53 $ bzr log -r last:
```

7.5.6 Diff between revisions

Diff between two revision numbers:

```
cburns@formats 19:53:18 $ bzr diff -r1516..1538 analyze.py
```

Pipe it into colordiff for colored output:

```
cburns@formats 19:53:18 $ bzr diff -r1516..1538 analyze.py | colordiff
```

7.6 Testing

Nipy uses the Numpy test framework which is based on nose. If you plan to do much development you should familiarize yourself with nose and read through the numpy testing guidelines.

7.6.1 Writing tests

Test files

The numpy testing framework and nipy extensions are imported with one line in your test module:

```
from nipy.testing import *
```

This imports all the `assert_*` functions you need like `assert_equal`, `assert_raises`, `assert_array_almost_equal` etc..., numpy's `rand` function, and the numpy test decorators: `knownfailure`, `slow`, `skipif`, etc...

Please name your test file with the *test_* prefix followed by the module name it tests. This makes it obvious for other developers which modules are tested, where to add tests, etc... An example test file and module pairing:

```
neuroimaging/core/reference/coordinate_system.py
neuroimaging/core/reference/tests/test_coordinate_system.py
```

All tests go in a test subdirectory for each package.

Temporary files

If you need to create a temporary file during your testing, you should use either of these two methods:

1. StringIO

   StringIO creates an in memory file-like object. The memory buffer is freed when the file is closed. This is the preferred method for temporary files in tests.

2. tempfile.mkstemp

   This will create a temporary file which can be used during testing. There are parameters for specifying the filename *prefix* and *suffix*.

   Note: The tempfile module includes a convenience function *NamedTemporaryFile* which deletes the file automatically when it is closed. However, whether the files can be opened a second time varies across platforms and there are problems using this function *Windows*.

Both of the above libraries are preferred over creating a file in the test directory and then removing them with a call to `os.remove`. For various reasons, sometimes `os.remove` doesn't get called and temp files get left around.

mkstemp example:

```
from tempfile import mkstemp
fd, name = mkstemp(suffix='.nii.gz')
tmpfile = open(name)
save_image(fake_image, tmpfile.name)
tmpfile.close()
os.unlink(name)  # This deletes the temp file
```

Many tests in one test function

To keep tests organized, it's best to have one test function correspond to one class method or module-level function. Often though, you need many individual tests to thoroughly cover (100% coverage) the method/function. This calls for a generator function. Use a `yield` statement to run each individual test, independent from the other tests. This prevents the case where the first test fails and as a result the following tests don't get run.

This test function executes four independent tests:

```
def test_index():
    cs = CoordinateSystem('ijk')
    yield assert_equal, cs.index('i'), 0
    yield assert_equal, cs.index('j'), 1
    yield assert_equal, cs.index('k'), 2
    yield assert_raises, ValueError, cs.index, 'x'
```

Suppress *warnings* on test output

In order to reduce noise when running the tests, consider suppressing *warnings* in your test modules. This can be done in the module-level setup and teardown functions:

```
import warnings
...

def setup():
    # Suppress warnings during tests to reduce noise
    warnings.simplefilter("ignore")
```

```
def teardown():
    # Clear list of warning filters
    warnings.resetwarnings()
```

7.6.2 Running tests

Running the full test suite

For our tests, we have collected a set of fmri imaging data which are required for the tests to run. The data must be downloaded separately and installed in your home directory `$HOME/.nipy/tests/data`. From your home directory:

```
mkdir -p .nipy/tests/data
svn co http://neuroimaging.scipy.org/svn/ni/data/trunk/fmri .nipy/tests/data
```

Tests can be run on the package:

```
import nipy as ni
ni.test()
```

Running individual tests

You can also run nose from the command line with a variety of options. To test an individual module:

```
nosetests test_image.py
```

To test an individual function:

```
nosetests test_module:test_function
```

To test a class:

```
nosetests test_module:TestClass
```

To test a class method:

```
nosetests test_module:TestClass.test_method
```

Verbose mode (*-v* option) will print out the function names as they are executed. Standard output is normally supressed by nose, to see any print statements you must include the *-s* option. In order to get a "full verbose" output, call nose like this:

```
nosetests -sv test_module.py
```

To include doctests in the nose test:

```
nosetests -sv --with-doctest test_module.py
```

For details on all the command line options:

```
nosetests --help
```

7.6.3 Coverage Testing

Coverage testing is a technique used to see how much of the code is exercised by the unit tests. It is important to remember that a high level of coverage is a necessary but not sufficient condition for having effective tests. Coverage testing can be useful for identifying whole functions or classes which are not tested, or for finding certain conditions which are never tested.

This is an excellent task for nose - the automated test runner we are using. Nose can run the python coverage tester. First make sure you have the coverage tester installed on your system. Download the tarball from the link, extract and install `python setup.py install`. Or on Ubuntu you can install from apt-get: `sudo apt-get install python-coverage`.

Run nose with coverage testing arguments:

```
nosetests -sv --with-coverage path_to_code
```

For example, this command:

```
nosetests -sv --with-coverage test_coordinate_map.py
```

will report the following:

```
Name                                                 Stmts   Exec  Cover   Missing
----------------------------------------------------------------------------------
neuroimaging                                            21     14    66%   70-74, 88-89
neuroimaging.core                                        4      4   100%
neuroimaging.core.reference                              8      8   100%
neuroimaging.core.reference.array_coords               100     90    90%   133-134, 148-151, 220, 222, 235
neuroimaging.core.reference.coordinate_map             188    187    99%   738
neuroimaging.core.reference.coordinate_system           61     61   100%
neuroimaging.core.reference.slices                      34     34   100%
neuroimaging.core.transforms                             0      0   100%
neuroimaging.core.transforms.affines                    14     14   100%
```

The coverage report will cover any python source module imported after the start of the test. This can be noisy and difficult to focus on the specific module for which you are writing nosetests. For instance, the above report also included coverage of most of `numpy`. To focus the coverage report, you can provide nose with the specific package you would like output from using the `--cover-package`. For example, in writing tests for the coordinate_map module:

```
nosetests --with-coverage --cover-package=neuroimaging.core.reference.coordinate_map test_coordinate_
```

Since that's a lot to type, I wrote a tool called `sneeze` to that simplifies coverage testing with nose.

Sneeze

Sneeze runs nose with coverage testing and reports only the package the test module is testing. It requires the test module follow a simple naming convention:

1. Prefix `test_`
2. The package name you are testing
3. Suffix `.py`

For example, the test module for the `coordinate_map` module is named `test_coordinate_map.py`. Then testing coverage is as simple as:

```
sneeze.py test_coordinate_map.py
```

Sneeze is included in the `tools` directory in the nipy source. Simply run the `setup.py` to install sneeze in your local bin directory.

7.7 Debugging

Some options are:

7.7.1 Run in ipython

As in:

```
In [1]: run mymodule.py
... (somecrash)
In [2]: %debug
```

Then diagnose, using the workspace that comes up, which has the context of the crash.

You can also do:

```
In [1] %pdb on
In [2]: run mymodule.py
... (somecrash)
```

At that point you will be automatically dropped into the the workspace in the context of the error. This is very similar to the matlab `dbstop if error` command.

See the ipython manual , and debugging in ipython for more detail.

7.7.2 Embed ipython in crashing code

Often it is not possible to run the code directly from ipython using the `run` command. For example, the code may be called from some other system such as sphinx. In that case you can embed. At the point that you want ipython to open with the context available for instrospection, add:

```
from IPython.Shell import IPShellEmbed
ipshell = IPShellEmbed()
ipshell()
```

See embedding ipython for more detail.

7.8 Optimization

In the early stages of NIPY development, we are focusing on functionality and usability. In regards to optimization, we benefit **significantly** from the optimized routines in scipy and numpy. As NIPY progresses it is likely we will spend more energy on optimizing critical functions. In our py4science group at UC Berkeley we've had several meetings on the various optimization options including ctypes, weave and blitz, and cython. It's clear there are many good options, including standard C-extensions. However, optimized code tends to be less readable and more difficult to

debug and maintain. When we do optimize our code we will first profile the code to determine the offending sections, then optimize those sections. Until that need arises, we will follow the great advice from these fellow programmers:

Kent Beck: "First make it work. Then make it right. Then make it fast."

Donald Knuth on optimization:

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

Tim Hochberg, from the Numpy list:

```
0. Think about your algorithm.
1. Vectorize your inner loop.
2. Eliminate temporaries
3. Ask for help
4. Recode in C.
5. Accept that your code will never be fast.

Step zero should probably be repeated after every other step ;)
```

7.9 Open Source Development

For those interested in more info about contributing to an open source project, Here are some links I've found. They are probably no better or worse than other similar documents:

- Software Release Practice HOWTO
- Contributing to Open Source Projects HOWTO

7.10 The ChangeLog

NOTE: We have not kepted up with our ChangeLog. This is here for future reference. We will be more diligent with this when we have regular software releases.

If you are a developer with commit access, **please** fill a proper ChangeLog entry per significant change. The SVN commit messages may be shorter (though a brief summary is appreciated), but a detailed ChangeLog is critical. It gives us a history of what has happened, allows us to write release notes at each new release, and is often the only way to backtrack on the rationale for a change (as the diff will only show the change, not **why** it happened).

Please skim the existing ChangeLog for an idea of the proper level of detail (you don't have to write a novel about a patch).

The existing ChangeLog is generated using (X)Emacs' fantastic ChangeLog mode: all you have to do is position the cursor in the function/method where the change was made, and hit 'C-x 4 a'. XEmacs automatically opens the ChangeLog file, mark a dated/named point, and creates an entry pre-titled with the file and function name. It doesn't get any better than this. If you are not using (X)Emacs, please try to follow the same convention so we have a readable, organized ChangeLog.

To get your name in the ChangeLog, set this in your .emacs file:

(setq user-full-name "Your Name") (setq user-mail-address "youradddress@domain.com")

Feel free to obfuscate or omit the address, but at least leave your name in. For user contributions, try to give credit by name on patches or significant ideas, but please do an @ -> -AT- replacement in the email addresses (users have asked for this in the past).

CHAPTER

EIGHT

DEVELOPMENT PLANNING

8.1 Nipy roadmap

We plan to release a protoype of NIPY by the Summer of 2009. This will include a full FMRI analysis, 2D visualization, and integration with other packages for spatial processing (SPM and FSL). We will continue to improve our documentation and tutorials with the aim of providing a full introduction to neuroimaging analysis.

We will also extend our collaborations with other neuroimaging groups, integrating more functionality into NIPY and providing better interoperability with other packages. This will include the design and implementation of a pipeline/batching system, integration of registration algorithms, and improved 2D and 3D visualization.

8.2 TODO for nipy development

This document will serve to organize current development work on nipy. It will include current sprint items, future feature ideas, and design discussions, etc...

8.2.1 Current Sprint

Goal for Feb 27

Cleanup and document the Image and CoordinateMap classes. There have been various changed to the CoordinateMap classes lately, merge Jonathan's branch, finish remaining changes, update docstrings and doctests. Write tutorials explaining these base classes.

Working prototype for interfacing with SPM.

Working prototype for registration visualization.

Goal for March 20

Review fff2.neuro code and prepare for sprint.

8.2.2 Documentation

- Create NIPY sidebar with links to all project related websites.
- Create a Best Practices document.
- Create a rst doc for *Request a review* process.

Tutorials

Tutorials are an excellent way to document and test the software. Some ideas for tutorials to write in our Sphinx documentation (in no specific order):

- Slice timing
- Image resampling
- Image IO
- Registration using SPM/FSL
- FMRI analysis
- Making one 4D image from many 3D images, and vice versa. Document ImageList and FmriImageList.
- Apply SPM registration .mat to a NIPY image.
- Create working example out of this TRAC pca page. Should also be a rest document.
- Add analysis pipeline(s) blueprint.

8.2.3 Bugs

These should be moved to the nipy bug section on launchpad. Placed here until they can be input.

- Fix fmri.pca module. Internally it's referencing old image api that no longer exists like Image.slice_iterator. Currently all tests are skipped or commented out.
- FmriStat has undefined objects, FmriStatOLS and FmriStatAR. Look into modalities.fmri.fmristat.tests.test_utils.py
- Fix possible precision error in fixes.scipy.ndimage.test_registration function test_autoalign_nmi_value_2. See FIXME.
- Fix error in test_segment test_texture2 functions (fixes.scipy.ndimage). See FIXME.
- import nipy.algorithms is very slow! Find and fix. The shared library is slow.
- base class for all new-style classes should be *object*

8.2.4 Data

- Replace fmri test file `funcfile` with a reasonable fmri file. It's shape is odd, (20,20,2,20). Many tests have been updated to this file and will need to me modified. `funcfile` is located in neuroimaging/testing/functinal.nii.gz

8.2.5 Refactorings

- Remove path.py and replace datasource with numpy's version. datasource and path.py cleanup should be done together as nipy's datasource is one of the main users of path.py:
  - Convert from nipy datasource to numpy datasource. Then remove nipy's datasource.py
  - Delete neuroimaging/utils/path.py. This custom path module does not provide any benefit over os.path. Using a non-standard path module adds confusion to the code. This will require going through the code base and updating all references to the path module. Perhaps a good use of grin for a global search and replace.

- Rewrite weave code in algorithms/statistics/intrinsic_volumes.py as C extension.
- Cleanup neuroimaging.testing directory. Possibly rename 'testing' to 'tests'. Move utils.tests.data.__init__.py to tests and update import statements in all test modules.
- image.save function should accept filename or file-like object. If I have an open file I would like to be able to pass that in also, instead of fp.name. Happens in test code a lot.
- image._open function should accept Image objects in addition to ndarrays and filenames. Currently the save function has to call np.asarray(img) to get the data array out of the image and pass them to _open in order to create the output image.
- Add dtype options when saving. When saving images it uses the native dtype for the system. Should be able to specify this. in the test_file_roundtrip, self.img is a uint8, but is saved to tmpfile as float64. Adding this would allow us to save images without the scaling being applied.
- In image._open(url, ...), should we test if the "url" is a PyNiftiIO object already? This was in the tests from 'old code' and passed:

  ```
  new = Image(self.img._data, self.img.grid)
  ```

  img._data is a PyNIftiIO object. It works, but we should verify it's harmless otherwise prevent it from happening.
- Look at image.merge_image function. Is it still needed? Does it fit into the current api?
- Automated test for modalities.fmri.pca, check for covariance diagonal structure, post pca.
- FmriImageList.emptycopy() - Is there a better way to do this? Matthew proposed possibly implementing Gael's dress/undress metadata example.
- Verify documentation of the image generators. Create a simple example using them.
- Use python 2.5 feature of being able to reset the generator?
- Add test data where volumes contain intensity ramps. Slice with generator and test ramp values.
- Implement fmriimagelist blueprint.

8.2.6 Code Design Thoughts

A central location to dump thoughts that could be shared by the developers and tracked easily.

8.2.7 Future Features

Put ideas here for features nipy should have but are not part of our current development. These features will eventually be added to a weekly sprint log.

- Auto backup script for nipy repos to run as weekly cron job. We should setup a machine to perform regular branch builds and tests. This would also provide an on-site backup.
- See if we can add bz2 support to nifticlib.
- Should image.load have an optional squeeze keyword to squeeze a 4D image with one frame into a 3D image?

CHAPTER

NINE

SOFTWARE DESIGN

9.1 Understanding voxel and real world mappings

9.1.1 Voxel coordinates and real-world coordinates

A point can be represented by coordinates relative to specified axes. coordinates are (almost always) numbers - see coordinate systems

For example, a map grid reference gives a coordinate (a pair of numbers) to a point on the map. The numbers give the respective positions on the horizontal (`x`) and vertical (`y`) axes of the map.

A coordinate system is defined by a set of axes. In the example above, the axes are the `x` and `y` axes. Axes for coordinates are usually orthogonal - for example, moving one unit up on the `x` axis on the map causes no change in the `y` coordinate - because the axes are at 90 degrees.

In this discussion we'll concentrate on the three dimensional case. Having three dimensions means that we have a three axis coordinate system, and coordinates have three values. The meaning of the values depend on what the axes are.

Voxel coordinates

Array indexing is one example of using a coordinate system. Let's say we have a three dimensional array:

```
A = np.arange(24).reshape((2,3,4))
```

The value `0` is at array coordinate `0,0,0`:

```
assert A[0,0,0] == 0
```

and the value `23` is at array coordinate `1,2,3`:

```
assert A[1,2,3] == 23
```

(remembering python's zero-based indexing). If we now say that our array is a 3D volume element array - an array of voxels, then the array coordinate is also a voxel coordinate.

If we want to use `numpy` to index our array, then we need integer voxel coordinates, but if we use a resampling scheme, we can also imagine non-integer voxel coordinates for `A`, such as `(0.6,1.2,1.9)`, and we could use resampling to estimate the value at such a coordinate, given the actual data in the surrounding (integer) points.

Array / voxel coordinates refer to the array axes. Without any further information, they do not tell us about where the point is in the real world - the world we can measure with a ruler. We refer to array / voxel coordinates with indices

`i, j, k`, where `i` is the first value in the 3 value coordinate tuple. For example, if array / voxel point `(1,2,3)` has `i=1, j=2, k=3`. We'll be careful only to use `i, j, k` rather than `x, y, z`, because we are going to use `x, y, z` to refer to real-world coordinates.

Real-world coordinates

Real-world coordinates are coordinates where the values refer to real-world axes. A real-world axis is an axis that refers to some real physical space, like low to high position in an MRI scanner, or the position in terms of the subject's head.

Here we'll use the usual neuroimaging convention, and that is to label our axes relative to the subject's head:

- `x` has negative values for left and positive values for right
- `y` has negative values for posterior (back of head) and positive values for anterior (front of head)
- `z` has negative values for the inferior (towards the neck) and postive values for superior (towards the highest point of the head, when standing)

9.1.2 Image index ordering

Background

In general, images - and in particular NIfTI format images, are ordered in memory with the X dimension changing fastest, and the Z dimension changing slowest.

Numpy has two different ways of indexing arrays in memory, C and fortran. With C index ordering, the first index into an array indexes the slowest changing dimension, and the last indexes the fastest changing dimension. With fortran ordering, the first index refers to the fastest changing dimension - X in the case of the image mentioned above.

C is the default index ordering for arrays in Numpy.

For example, let's imagine that we have a binary block of 3D image data, in standard NIfTI / Analyze format, with the X dimension changing fastest, called *my.img*, containing Float32 data. Then we memory map it:

```
img_arr = memmap('my.img', dtype=float32)
```

When we index this new array, the first index indexes the Z dimension, and the third indexes X. For example, if I want a voxel X=3, Y=10, Z=20 (zero-based), I have to get this from the array with:

```
img_arr[20, 10, 3]
```

The problem

Most potential users of NiPy are likely to have experience of using image arrays in Matlab and SPM. Matlab uses Fortran index ordering. For fortran, the first index is the fastest changing, and the last is the slowest-changing. For example, here is how to get voxel X=3, Y=10, Z=20 (zero-based) using SPM in Matlab:

```
img_arr = spm_read_vols(spm_vol('my.img'));
img_arr(4, 11, 21)  % matlab indexing is one-based
```

This ordering fits better with the way that we talk about coordinates in functional imaging, as we invariably use XYZ ordered coordinates in papers. It is possible to do the same in numpy, by specifying that the image should have fortran index ordering:

```
img_arr = memmap('my.img', dtype=float32, order='F')
img_arr[3, 10, 20]
```

The proposal

Change the default ordering of image arrays to fortran, in order to allow XYZ index ordering. So, change the access to the image array in the image class so that, to get the voxel at X=3, Y=10, Z=20 (zero-based):

```
img = Image('my.img')
img[3, 10, 20]
```

instead of the current situation, which requires:

```
img = Image('my.img')
img[20, 10, 3]
```

Summary of discussion

For:

- Fortran index ordering is more intuitive for functional imaging because of conventional XYZ ordering of spatial coordinates, and Fortran index ordering in packages such as Matlab
- Indexing into a raw array is fast, and common in lower-level applications, so it would be useful to implement the more intuitive XYZ ordering at this level rather than via interpolators (see below)
- Standardizing to one index ordering (XYZ) would mean users would not have to think about the arrangement of the image in memory

Against:

- C index ordering is more familiar to C users
- C index ordering is the default in numpy
- XYZ ordering can be implemented by wrapping by an interpolator

Potential problems

Performance penalties

KY commented:

```
This seems like a good idea to me but I have no knowledge of numpy
internals (and even less than none after the numeric/numarray
integration). Does anyone know if this will (or definitely will not)
incur any kind of obvious performance penalties re. array operations
(sans arcane problems like stride issues in huge arrays)?
```

MB replied:

> Note that, we are not proposing to change the memory layout of the image, which is fixed by the image format in e.g NIfTI, but only to index it XYZ instead of ZYX. As far as I am aware, there are no significant performance differences between:

```
img_arr = memmap('my.img', dtype=float32, order='C')
img_arr[5,4,3]
```

and:

```
img_arr = memmap('my.img', dtype=float32, order='F')
img_arr[3,4,5]
```

Happy to be corrected though.

Clash between default ordering of numpy arrays and nipy images

C index ordering is the default in numpy, and using fortran ordering for images might be confusing in some circumstances. Consider for example:

> img_obj = Image('my.img') # Where the Image class has been changed to implement Fortran ordering
> first_z_slice = img_obj[...,0] # returns a Z slice
>
> img_arr = memmap('my.img', dtype=float32) # C ordering, the numpy default img_obj = Image(img_arr)
> first_z_slice = img_obj[...,0] # in fact returns an X slice

I suppose that we could check that arrays are fortran index ordered in the Image __init__ routine.

An alternative proposal - XYZ ordering of output coordinates

JT: Another thought, that is a compromise between the XYZ coordinates and Fortran ordering.

To me, having worked mostly with C-type arrays, when I index an array I think in C terms. But, the Image objects have the "warp" attached to them, which describes the output coordinates. We could insist that the output coordinates are XYZT (or make this an option). So, for instance, if the 4x4 transform was the identity, the following two calls would give something like:

```
interp = interpolator(img)
img[3,4,5] == interp(5,4,3)
```

This way, users would be sure in the interpolator of the order of the coordinates, but users who want access to the array would know that they would be using the array order on disk...

I see that a lot of users will want to think of the first coordinate as "x", but depending on the sampling the [0] slice of img may be the leftmost or the rightmost. To find out which is which, users will have to look at the 4x4 transform (or equivalently the start and the step). So just knowing the first array coordinate is the "x" coordinate still misses some information, all of which is contained in the transform.

MB replied:

I agree that the output coordinates are very important - and I think we all agree that this should be XYZ(T)?

For the raw array indices - it is very common for people to want to do things to the raw image array - the quickstart examples containing a few - and you usually don't care about which end of X is left in that situation, only which spatial etc dimension the index refers to.

9.2 Image index ordering

9.2.1 Background

In general, images - and in particular NIfTI format images, are ordered in memory with the X dimension changing fastest, and the Z dimension changing slowest.

Numpy has two different ways of indexing arrays in memory, C and fortran. With C index ordering, the first index into an array indexes the slowest changing dimension, and the last indexes the fastest changing dimension. With fortran ordering, the first index refers to the fastest changing dimension - X in the case of the image mentioned above.

C is the default index ordering for arrays in Numpy.

For example, let's imagine that we have a binary block of 3D image data, in standard NIfTI / Analyze format, with the X dimension changing fastest, called *my.img*, containing Float32 data. Then we memory map it:

```
img_arr = memmap('my.img', dtype=float32)
```

When we index this new array, the first index indexes the Z dimension, and the third indexes X. For example, if I want a voxel X=3, Y=10, Z=20 (zero-based), I have to get this from the array with:

```
img_arr[20, 10, 3]
```

9.2.2 The problem

Most potential users of NiPy are likely to have experience of using image arrays in Matlab and SPM. Matlab uses Fortran index ordering. For fortran, the first index is the fastest changing, and the last is the slowest-changing. For example, here is how to get voxel X=3, Y=10, Z=20 (zero-based) using SPM in Matlab:

```
img_arr = spm_read_vols(spm_vol('my.img'));
img_arr(4, 11, 21)  % matlab indexing is one-based
```

This ordering fits better with the way that we talk about coordinates in functional imaging, as we invariably use XYZ ordered coordinates in papers. It is possible to do the same in numpy, by specifying that the image should have fortran index ordering:

```
img_arr = memmap('my.img', dtype=float32, order='F')
img_arr[3, 10, 20]
```

9.2.3 Native fortran or C indexing for images

We could change the default ordering of image arrays to fortran, in order to allow XYZ index ordering. So, change the access to the image array in the image class so that, to get the voxel at X=3, Y=10, Z=20 (zero-based):

```
img = load_image('my.img')
img[3, 10, 20]
```

instead of the current situation, which requires:

```
img = load_image('my.img')
img[20, 10, 3]
```

For and against fortran ordering

For:

- Fortran index ordering is more intuitive for functional imaging because of conventional XYZ ordering of spatial coordinates, and Fortran index ordering in packages such as Matlab
- Indexing into a raw array is fast, and common in lower-level applications, so it would be useful to implement the more intuitive XYZ ordering at this level rather than via interpolators (see below)
- Standardizing to one index ordering (XYZ) would mean users would not have to think about the arrangement of the image in memory

Against:

- C index ordering is more familiar to C users
- C index ordering is the default in numpy
- XYZ ordering can be implemented by wrapping by an interpolator

Note that there is no performance penalty for either array ordering, as this is dealt with internally by NumPy. For example, imagine the following:

```
arr = np.empty((100,50)) # Indexing is C by default
arr2 = arr.transpose() # Now it is fortran
# There should be no effective difference in speed for the next two lines
b = arr[0] # get first row of data - most discontiguous memory
c = arr2[:,0] # gets same data, again most discontiguous memory
```

Potential problems for fortran ordering

Clash between default ordering of numpy arrays and nipy images

C index ordering is the default in numpy, and using fortran ordering for images might be confusing in some circumstances. Consider for example:

```
img_obj = load_image('my.img') # Where the Image class has been changed to implement Fortran ordering
first_z_slice = img_obj[...,0] # returns a Z slice

img_arr = memmap('my.img', dtype=float32) # C ordering, the numpy default
img_obj = Image.from_array(img_arr) # this call may not be correct
first_z_slice = img_obj[...,0]  # in fact returns an X slice
```

I suppose that we could check that arrays are fortran index ordered in the Image __init__ routine.

9.2.4 An alternative proposal - XYZ ordering of output coordinates

JT: Another thought, that is a compromise between the XYZ coordinates and Fortran ordering.

To me, having worked mostly with C-type arrays, when I index an array I think in C terms. But, the Image objects have the "warp" attached to them, which describes the output coordinates. We could insist that the output coordinates are XYZT (or make this an option). So, for instance, if the 4x4 transform was the identity, the following two calls would give something like:

```
 >>> interp = interpolator(img)
 >>> img[3,4,5] == interp(5,4,3)
True
```

This way, users would be sure in the interpolator of the order of the coordinates, but users who want access to the array would know that they would be using the array order on disk...

I see that a lot of users will want to think of the first coordinate as "x", but depending on the sampling the [0] slice of img may be the leftmost or the rightmost. To find out which is which, users will have to look at the 4x4 transform (or equivalently the start and the step). So just knowing the first array coordinate is the "x" coordinate still misses some information, all of which is contained in the transform.

MB replied:

I agree that the output coordinates are very important - and I think we all agree that this should be XYZ(T)?

For the raw array indices - it is very common for people to want to do things to the raw image array - the quickstart examples containing a few - and you usually don't care about which end of X is left in that situation, only which spatial etc dimension the index refers to.

9.3 Registration API Design

This contains design ideas for the end-user api when registering images in nipy.

We want to provide a simple api, but with enough flexibility to allow users to changes various components of the pipeline. We will also provide various **Standard** scripts that perform typical pipelines.

The pluggable script:

```
func_img = load_image(filename)
anat_img = load_image(filename)
interpolator = SplineInterpolator(order=3)
metric = NormalizedMutualInformation()
optimizer = Powell()
strategy = RegistrationStrategy(interpolator, metric, optimizer)
w2w = strategy.apply(img_fixed, img_moving)
```

To apply the transform and resample the image:

```
new_img = resample(img_moving, w2w, interp=interpolator)
```

Or:

```
new_img = Image(img_moving, w2w*img_moving.coordmap)
```

9.3.1 Transform Multiplication

The multiplication order is important and coordinate systems must *make sense*. The *output coordinates* of the mapping on the right-hand of the operator, must match the *input coordinates* of the mapping on the left-hand side of the operator.

For example, imageA has a mapping from voxels-to-world (v2w), imageB has a mapping from world-to-world (w2w). So the output of imageA, *world*, maps to the input of imageB, *world*. We would compose a new mapping (transform) from these mappings like this:

```
new_coordmap = imageB.coordmap * imageA.coordmap
```

If one tried to compose a mapping in the other order, an error should be raised as the code would detect a mismatch of trying to map output coordinates from imageB, *world* to the input coordinates of imageA, *voxels*:

```
new_coordmap = imageA.coordmap * imageB.coordmap
raise ValueError!!!
```

Note: We should consider a meaningful error message to help people quickly correct this mistake.

One way to remember this ordering is to think of composing functions. If these were functions, the output of the first function to evaluate (imageA.coordmap) is passed as input to the second function (imageB.coordmap). And therefore they must match:

```
new_coordmap = imageB.coordmap(imageA.coordmap())
```

9.3.2 Matching Coordinate Systems

We need to make sure we can detect mismatched coordinate mappings. The CoordinateSystem class has a check for equality (__eq__ method) based on the axis and name attributes. Long-term this may not be robust enough, but it's a starting place. We should write tests for failing cases of this, if they don't already exists.

9.3.3 CoordinateMap

Recall the CoordinateMap defines a mapping between two coordinate systems, an input coordinate system and an output coordinate system. One example of this would be a mapping from voxel space to scanner space. In a Nifti1 header we would have an affine transform to apply this mapping. The *input coordinates* would be voxel space, the *output coordinates* would be world space, and the affine transform provides the mapping between them.

9.4 Repository design

See also *Repository API* and *Can NIPY get something interesting from BrainVISA databases?*

For the NIPY system, there seems to be interest for the following:

- Easy distributed computing
- Easy scripting, replicating the same analysis on different data
- Flexibility - easy of inter-operation with other brain imaging systems

At a minimum, this seems to entail the following requirements for the NIPY repository system:

- Unique identifiers of data, which can be abstracted from the most local or convenient data storage
- A mechanism for mapping the canonical data model(s) from NIPY to an arbitrary, and potentially even inconsistent repository structure
- **A set of semantic primitives / metadata slots, enabling for example:**
  - "all scans from this subject"
  - "the first scan from every subject in the control group"
  - "V1 localizer scans from all subjects"

  - "Extract the average timecourse for each subject from the ROI defined by all voxels with t > 0.005 in the V1 localizer scan for that subject"

These problems are not unique to the problem of brain imaging data, and in many cases have been treated in the domains of database design, geospatial and space telescope data, and the semantic web. Technologies of particular interest include:

- HDF5 - the basis of MINC 2.0 (and potentially NIFTII 2), the most recent development in the more general CDF / HDF series (and very highly regarded). There are excellent python binding available in PyTables.
- Relational database design - it would be nice to efficiently select data based on any arbitrary subset of attributes associated with that data.
- The notion of URI developed under the guidance of the w3c. Briefly, a URI consists of:
  - An authority (i.e. a domain name controlled by a particular entity)
  - A path - a particular resource specified by that authority
  - Abstraction from storage (as opposed to a URL) - a URI does not necessarily include the information necessary for retrieving the data referred to, though it may.
- Ways of dealing with hierarchical data as developed in the XML field (though these strategies could be implemented potentially in other hierarchical data formats - even filesystems).

Note that incorporation of any of the above ideas does not require the use of the actual technology referenced. For example, relational queries can be made in PyTables in many cases **more efficiently** than in a relational database by storing everything in a single denormalized table. This data structure tends to be more efficient than the equivalent normalized relational database format in the cases where a single data field is much larger than the others (as is the case with the data array in brain imaging data). That said, adherance to standards allows us to leverage existing code which may be tuned to a degree that would be beyond the scope of this project (for example, fast Xpath query libraries, as made available via lxml in Python).

9.5 Can NIPY get something interesting from BrainVISA databases?

I wrote this document to try to give more information to the NIPY developers about the present and future of *BrainVISA* database system. I hope it will serve the discussion opened by Jarrod Millman about a possible collaboration between the two projects on this topic. Unfortunately, I do not know other projects providing similar features (such as BIRN) so I will only focus on BrainVISA.

Yann Cointepas

2006-11-21

9.5.1 Introduction

In BrainVISA, all the database system is home made and written in Python. This system is based on the file system and allows to do requests for both reading and writing (get the name of non existing files). We will change this in the future by defining an API (such the one introduced below) and by using at least two implementations, one relying on a relational database system and one compatible with the actual database system. Having one single API will make it possible, for instance, to work on huge databases located on servers and on smaller databases located in a laptop directory (with some synchronization features). This system will be independent from the BrainVISA application, it could be packaged separately. Unfortunately, we cannot say when this work will be done (our developments are slowed because all our lab will move in a new institute in January 2007). Here is a summary describing actual BrainVISA database system and some thoughts of what it may become.

9.5.2 What is a database in BrainVISA today?

A directory is a BrainVISA database if the structure of its sub-directories and the file names in this directory respect a set of rules. These rules make it possible to BrainVISA to scan the whole directory contents and to identify without ambiguity the database elements. These elements are composed of the following information:

- ***Data type:*** **identify the contents of a data (image, mesh,** functional image, anatomical RM, etc). The data types are organized in hierarchy making it possible to decline a generic type in several specialized types. For example, there is a 4D Image type which is specialized in 3D Image. 3D Image is itself declined in several types of which T1 MRI and Brain mask.
- ***File format:*** **Represent the format of files used to record a** data. BrainVISA is able to recognize several file formats (for example DICOM, Analyze/SPM, GIS, etc). It is easy to add new data formats and to provide converters to make it possible for existing processes to use these new formats.
- ***Files:*** **contains the names of the files (and/or directories) used** to record the data.
- ***Attributes:*** **an attribute is an association between a name and a** value. A set of attributes is associated with each element of BrainVISA database. This set represents all of the characteristics of a data (as the image size, the name of the protocol corresponding to the data or the acquisition parameters). Attributes values are set by BrainVISA during directory scanning (typically protocol, group, subject, etc.).

It is possible to completely define the set of rules used to convert a directory in a BrainVISA database. That allows the use of BrainVISA without having to modify an existing file organization. However, the writing of such a system of rules requires very good knowledge of BrainVISA. This is why BrainVISA is provided with a default data organization system that can be used easily.

A database can be used for deciding where to write data. The set of rules is used to generate the appropriate file name according to the data type, file format and attributes. This is a key feature that greatly helps the users and allow automation.

It is not mandatory to use a database to process data with BrainVISA. However, some important features are not available when you are using data which are not in a database. For example, the BrainVISA ability to construct a default output file name when an input data is selected in a process relies on the database system. Moreover, some processes use the database system to find data; for example, the brain mask viewer tries to find the T1 MRI used to build the brain mask in order to superimpose both images in an Anatomist window.

9.5.3 A few thoughts about a possible API for repositories

I think the most important point for data repositories is to define an user API. This API should be independent of data storage and of data organization. Data organization is important because it is very difficult to find a single organization that covers the needs of all users in the long term. In this API, each data item should have an unique identifier (let's call it an URL). The rest of the API could be divided in two parts:

1. An indexation system managing data organization. It defines properties attached to data items (for instance, "group" or "subject" can be seen as properties of an FMRI image) as well as possible user requests on the data. This indexation API could have several implementations (relational database, BIRN, BrainVISA, etc.).
2. A data storage system managing the link between the URL of a data item and its representation on a local file system. This system should take into account various file formats and various file storage systems (e.g. on a local file system, on a distant ftp site, as bytes blocks in a relational database).

This separation between indexation and storage is important for the design of databases, it makes it possible, for instance, to use distant or local data storage, or to define several indexations (i.e. several data organizations) for the same data. However indexation and data storage are not always independent. For example, they are independent if we use a relational database for indexation and URLs for storage, but they are not if file or directory names give indexation information (like in BrainVISA databases described above). At the user level, things can be simpler because the separation can be hidden in one object: the repository. A repository is composed of one indexation system and one

data storage system and manage all the links between them. The user can send requests to the repository and receive a set of data items. Each data item contains indexation information (via the indexation system) and gives access to the data (via the storage system). Here is a sample of what-user-code-could-be to illustrate what I have in mind followed by a few comments:

```
# Get an acces to one repository
repository = openRepository( repositoryURL )
# Create a request for selection of all the FMRI in the repository
request = 'SELECT * FROM FMRI'
# Iterate on data items in the repository
for item in repository.select( request ):
  print item.url
  # Item is a directory-like structure for properties access
  for property in item:
    print property, '=', item[ property ]
  # Retrieve the file(s) (and directorie(s) if any) from the data storage system
  # and convert it to NIFTI format (if necessary).
  files = item.getLocalFiles( format='NIFTI' )
  niftiFileName = files[ 0 ]
  # Read the image and do something with it
  ...
```

1. I do not yet have a good idea of how to represent requests. Here, I chose to use SQL since it is simple to understand.
2. This code does not make any assumption on the properties that are associated to an FMRI image.
3. The method getLocalFiles can do nothing more than return a file name if the data item correspond to a local file in NIFTI format. But the same code can be used to acces a DICOM image located in a distant ftp server. In this case, getLocalFiles will manage the transfer of the DICOM file, then the conversion to the required NIFTI format and return name of temporary file(s).
4. getLocalFiles cannot always return just one file name because on the long term, there will be many data types (FMRI, diffusion MRI, EEG, MEG, etc.) that are going to be stored in the repositories. These different data will use various file formats. Some of these formats can use a combination of files and directories (for instance, CTF MEG raw data are stored in a directory (`*.ds`), the structural sulci format of BrainVISA is composed of a file(`*.arg`) and a directory (`*.data`), NIFTI images can be in one or two files, etc.).
5. The same kind of API can be used for writing data items in a repository. One could build a data item, adds properties and files and call something like repository.update(item).

9.6 Repository API

See also *Repository design* and *Can NIPY get something interesting from BrainVISA databases?*

FMRI datasets often have the structure:

- Group (sometimes) e.g. Patients, Controls
  - Subject e.g. Subject1, Subject2
    - Session e.g. Sess1, Sess1

How about an interface like:

```
repo = GSSRespository(
    root_dir = '/home/me/data/experiment1',
    groups = {'patients':
```

```
                {'subjects':
                 {'patient1':
                  {'sess1':
                   'filter': 'raw*nii'},
                  {'sess2':
                   'filter': 'raw*nii'}
                  },
                 {'patient2':
                  {'sess1':
                   'filter': 'raw*nii'}
                  {'sess2':
                   'filter': 'raw*nii'}
                  }
                 },
                'controls':
                {'subjects':
                 {'control1':
                  {'sess1':
                   'filter': 'raw*nii'},
                  {'sess2':
                   'filter': 'raw*nii'}
                  },
                 {'control2':
                  {'sess1':
                   'filter': 'raw*nii'}
                  {'sess2':
                   'filter': 'raw*nii'}
                  }
                 }
                })

for group in repo.groups:
    for subject in group.subjects:
        for session in subject.sessions:
            img = session.image
            # do something with image
```

We would need to think about adding metadata such as behavioral data from the scanning session, and so on. I suppose this will help us move transparently to using something like HDF5 for data storage.

9.7 What would pipelining look like?

Imagine a repository that is a modified version of the one in *Repository API*

Then:

```
my_repo = SubjectRepository('/some/structured/file/system')
my_designmaker = MyDesignParser() # Takes parameters from subject to create design
my_pipeline = Pipeline([
   realignerfactory('fsl'),
   slicetimerfactory('nipy', 'linear'),
   coregisterfactory('fsl', 'flirt'),
   normalizerfactory('spm'),
   filterfactory('nipy', 'smooth', 8),
   designfactory('nipy', my_designmaker),
   ])
```

```
my_analysis = SubjectAnalysis(my_repo, subject_pipeline=my_pipeline)
my_analysis.do()
my_analysis.archive()
```

9.8 Defining use cases

9.8.1 Transformation use cases

Use cases for defining and using transforms on images.

We should be very careful to only use the terms `x, y, z` to refer to physical space. For voxels, we should use `i, j, k`, or `i', j', k'` (i prime, j prime k prime).

I have an image *Img*.

Image Orientation

I would like to know what the voxel sizes are.

I would like to determine whether it was acquired axially, coronally or sagittally. What is the brain orientation in relation to the voxels? Has it been acquired at an oblique angle? What are the voxel dimensions?:

```
img = load_image(file)
cm = img.coordmap
print cm

input_coords axis_i:
             axis_j:
             axis_k:

             effective pixel dimensions
                           axis_i: 4mm
                           axis_j: 2mm
                           axis_k: 2mm

input/output mapping
               <Affine Matrix>




                   x   y   z
                 ------------
               i|  90  90   0
               j|  90   0  90
               k| 180  90  90

               input axis_i maps exactly to output axis_z
               input axis_j maps exactly to output axis_y
               input axis_k maps exactly to output axis_x flipped 180

output_coords axis0: Left -> Right
              axis1: Posterior -> Anterior
              axis2: Inferior -> Superior
```

In the case of a mapping that does not exactly align the input and output axes, something like:

```
...
input/output mapping
               <Affine Matrix>

               input axis0 maps closest to output axis2
               input axis1 maps closest to output axis1
               input axis2 maps closest to output axis0
...
```

If the best matching axis is reversed compared to input axis:

```
...
input axis0 maps [closest|exactly] to negative output axis2
```

and so on.

Creating transformations / co-ordinate maps

I have an array *pixelarray* that represents voxels in an image and have a matrix/transform *mat* which represents the relation between the voxel coordinates and the coordinates in scanner space (world coordinates). I want to associate the array with the matrix:

```
img = load_image(infile)
pixelarray = np.asarray(img)
```

(*pixelarray* is an array and does not have a coordinate map.):

```
pixelarray.shape
(40,256,256)
```

So, now I have some arbitrary transformation matrix:

```
mat = np.zeros((4,4))
mat[0,2] = 2 # giving x mm scaling
mat[1,1] = 2 # giving y mm scaling
mat[2,0] = 4 # giving z mm scaling
mat[3,3] = 1 # because it must be so
# Note inverse diagonal for zyx->xyz coordinate flip
```

I want to make an `Image` with these two:

```
coordmap = voxel2mm(pixelarray.shape, mat)
img = Image(pixelarray, coordmap)
```

The `voxel2mm` function allows separation of the image *array* from the size of the array, e.g.:

```
coordmap = voxel2mm((40,256,256), mat)
```

We could have another way of constructing image which allows passing of *mat* directly:

```
img = Image(pixelarray, mat=mat)
```

or:

```
img = Image.from_data_and_mat(pixelarray, mat)
```

but there should be "only one (obvious) way to do it".

Composing transforms

I have two images, *img1* and *img2*. Each image has a voxel-to-world transform associated with it. (The "world" for these two transforms could be similar or even identical in the case of an fmri series.) I would like to get from voxel coordinates in *img1* to voxel coordinates in *img2*, for resampling:

```
imgA = load_image(infile_A)
vx2mmA = imgA.coordmap
imgB = load_image(infile_B)
vx2mmB = imgB.coordmap
mm2vxB = vx2mmB.inverse
# I want to first apply transform implied in
# cmA, then the inverse of transform implied in
# cmB.  If these are matrices then this would be
# np.dot(mm2vxB, vx2mmA)
voxA_to_voxB = mm2vxB.composewith(vx2mmA)
```

The (matrix) multiply version of this syntax would be:

```
voxA_to_voxB = mm2vxB * vx2mmA
```

Composition should be of form `Second.composewith(First)` - as in `voxA_to_voxB = mm2vxB.composewith(vx2mmA)` above. The alternative is `First.composewith(Second)`, as in `voxA_to_voxB = vx2mmA.composewith(mm2vxB)`. We choose `Second.composewith(First)` on the basis that people need to understand the mathematics of function composition to some degree - see wikipedia_function_composition.

Real world to real world transform

We remind each other that a mapping is a function (callable) that takes coordinates as input and returns coordinates as output. So, if *M* is a mapping then:

```
[i',j',k'] = M(i, j, k)
```

where the *i, j, k* tuple is a coordinate, and the *i', j', k'* tuple is a transformed coordinate.

Let us imagine we have somehow come by a mapping *T* that relates a coordinate in a world space (mm) to other coordinates in a world space. A registration may return such a real-world to real-world mapping. Let us say that *V* is a useful mapping matching the voxel coordinates in *img1* to voxel coordinates in *img2*. If *img1* has a voxel to mm mapping *M1* and *img2* has a mm to voxel mapping of *inv_M2*, as in the previous example (repeated here):

```
imgA = load_image(infile_A)
vx2mmA = imgA.coordmap
imgB = load_image(infile_B)
vx2mmB = imgB.coordmap
mm2vxB = vx2mmB.inverse
```

then the registration may return the some coordinate map, *T* such that the intended mapping *V* from voxels in *img1* to voxels in *img2* is:

```
mm2vxB_map = mm2vxB.mapping
vx2mmA_map = vx2mmA.mapping
V = mm2vxB_map.composewith(T.composedwith(vx2mmA_map))
```

To support this, there should be a CoordinateMap constructor that looks like this:

```
T_coordmap = mm2mm(T)
```

where *T* is a mapping, so that:

```
V_coordmap = mm2vxB.composewith(T_coordmap.composedwith(vx2mmA))
```

I have done a coregistration between two images, *img1* and *img2*. This has given me a voxel-to-voxel transformation and I want to store this transformation in such a way that I can use this transform to resample *img1* to *img2*. *Resampling use cases*

I have done a coregistration between two images, *img1* and *img2*. I may want this to give me a worldA-to-worldB transformation, where worldA is the world of voxel-to-world for *img1*, and worldB is the world of voxel-to-world of *img2*.

My *img1* has a voxel to world transformation. This transformation may (for example) have come from the scanner that acquired the image - so telling me how the voxel positions in *img1* correspond to physical coordinates in terms of the magnet isocenter and millimeters in terms of the primary gradient orientations (x, y and z). I have the same for *img2*. For example, I might choose to display this image resampled so each voxel is a 1mm cube.

Now I have these transformations: ST(*img1*-V2W), and ST(*img2*-V2W) (where ST is *scanner tranform* as above, and *V2W* is voxel to world).

I have now done a coregistration between *img1* and *img2* (somehow) - giving me, in addition to *img1* and *img2*, a transformation that registers *img1* and *img2*. Let's call this tranformation V2V(*img1*, *img2*), where V2V is voxel-to-voxel.

In actuality *img2* can be an array of images, such as series of fMRI images and I want to align all the *img2* series to *img1* and then take these voxel-to-voxel aligned images (the *img1* and *img2* array) and remap them to the world space (voxel-to-world). Since remapping is an interpolation operation I can generate errors in the resampled pixel values. If I do more than one resampling, error will accumulate. I want to do only a single resampling. To avoid the errors associated with resampling I will build a *composite transformation* that will chain the separate voxel-to-voxel and voxel-to-world transformations into a single transformation function (such as an affine matrix that is the result of multiplying the several affine matrices together). With this single *composite transformatio* I now resample *img1* and *img2* and put them into the world coordinate system from which I can make measurements.

9.8.2 Image model use cases

In which we lay out the various things that users and developers may want to do to images. See also *Resampling use cases*

I have a nifti image on disk and would like to load it and view it's header.

9.8.3 Resampling use cases

Use cases for image resampling. See also *Image model use cases*.

9.8.4 Batching use cases

Using the nipy framework for creating scripts to process whole datasets, for example movement correction, coregistration of functional to structural (intermodality), smoothing, statistics, inference.

CHAPTER

TEN

DEFINING USE CASES

10.1 Transformation use cases

Use cases for defining and using transforms on images.

We should be very careful to only use the terms `x, y, z` to refer to physical space. For voxels, we should use `i, j, k`, or `i', j', k'` (i prime, j prime k prime).

I have an image *Img*.

10.1.1 Image Orientation

I would like to know what the voxel sizes are.

I would like to determine whether it was acquired axially, coronally or sagittally. What is the brain orientation in relation to the voxels? Has it been acquired at an oblique angle? What are the voxel dimensions?:

```
img = load_image(file)
cm = img.coordmap
print cm

input_coords axis_i:
             axis_j:
             axis_k:

             effective pixel dimensions
                           axis_i: 4mm
                           axis_j: 2mm
                           axis_k: 2mm

input/output mapping
               <Affine Matrix>



                   x   y   z
                 ------------
               i|  90  90   0
               j|  90   0  90
               k| 180  90  90
```

```
               input axis_i maps exactly to output axis_z
               input axis_j maps exactly to output axis_y
               input axis_k maps exactly to output axis_x flipped 180

output_coords axis0: Left -> Right
              axis1: Posterior -> Anterior
              axis2: Inferior -> Superior
```

In the case of a mapping that does not exactly align the input and output axes, something like:

```
...
input/output mapping
               <Affine Matrix>

               input axis0 maps closest to output axis2
               input axis1 maps closest to output axis1
               input axis2 maps closest to output axis0
...
```

If the best matching axis is reversed compared to input axis:

```
...
input axis0 maps [closest|exactly] to negative output axis2
```

and so on.

10.1.2 Creating transformations / co-ordinate maps

I have an array *pixelarray* that represents voxels in an image and have a matrix/transform *mat* which represents the relation between the voxel coordinates and the coordinates in scanner space (world coordinates). I want to associate the array with the matrix:

```
img = load_image(infile)
pixelarray = np.asarray(img)
```

(*pixelarray* is an array and does not have a coordinate map.):

```
pixelarray.shape
(40,256,256)
```

So, now I have some arbitrary transformation matrix:

```
mat = np.zeros((4,4))
mat[0,2] = 2 # giving x mm scaling
mat[1,1] = 2 # giving y mm scaling
mat[2,0] = 4 # giving z mm scaling
mat[3,3] = 1 # because it must be so
# Note inverse diagonal for zyx->xyz coordinate flip
```

I want to make an `Image` with these two:

```
coordmap = voxel2mm(pixelarray.shape, mat)
img = Image(pixelarray, coordmap)
```

The `voxel2mm` function allows separation of the image *array* from the size of the array, e.g.:

```
coordmap = voxel2mm((40,256,256), mat)
```

We could have another way of constructing image which allows passing of *mat* directly:

```
img = Image(pixelarray, mat=mat)
```

or:

```
img = Image.from_data_and_mat(pixelarray, mat)
```

but there should be "only one (obvious) way to do it".

Composing transforms

I have two images, *img1* and *img2*. Each image has a voxel-to-world transform associated with it. (The "world" for these two transforms could be similar or even identical in the case of an fmri series.) I would like to get from voxel coordinates in *img1* to voxel coordinates in *img2*, for resampling:

```
imgA = load_image(infile_A)
vx2mmA = imgA.coordmap
imgB = load_image(infile_B)
vx2mmB = imgB.coordmap
mm2vxB = vx2mmB.inverse
# I want to first apply transform implied in
# cmA, then the inverse of transform implied in
# cmB.  If these are matrices then this would be
# np.dot(mm2vxB, vx2mmA)
voxA_to_voxB = mm2vxB.composewith(vx2mmA)
```

The (matrix) multiply version of this syntax would be:

```
voxA_to_voxB = mm2vxB * vx2mmA
```

Composition should be of form `Second.composewith(First)` - as in `voxA_to_voxB = mm2vxB.composewith(vx2mmA)` above. The alternative is `First.composewith(Second)`, as in `voxA_to_voxB = vx2mmA.composewith(mm2vxB)`. We choose `Second.composewith(First)` on the basis that people need to understand the mathematics of function composition to some degree - see wikipedia_function_composition.

Real world to real world transform

We remind each other that a mapping is a function (callable) that takes coordinates as input and returns coordinates as output. So, if *M* is a mapping then:

```
[i',j',k'] = M(i, j, k)
```

where the *i, j, k* tuple is a coordinate, and the *i', j', k'* tuple is a transformed coordinate.

Let us imagine we have somehow come by a mapping *T* that relates a coordinate in a world space (mm) to other coordinates in a world space. A registration may return such a real-world to real-world mapping. Let us say that *V* is a useful mapping matching the voxel coordinates in *img1* to voxel coordinates in *img2*. If *img1* has a voxel to mm mapping *M1* and *img2* has a mm to voxel mapping of *inv_M2*, as in the previous example (repeated here):

```
imgA = load_image(infile_A)
vx2mmA = imgA.coordmap
imgB = load_image(infile_B)
vx2mmB = imgB.coordmap
mm2vxB = vx2mmB.inverse
```

then the registration may return the some coordinate map, *T* such that the intended mapping *V* from voxels in *img1* to voxels in *img2* is:

```
mm2vxB_map = mm2vxB.mapping
vx2mmA_map = vx2mmA.mapping
V = mm2vxB_map.composewith(T.composedwith(vx2mmA_map))
```

To support this, there should be a CoordinateMap constructor that looks like this:

```
T_coordmap = mm2mm(T)
```

where *T* is a mapping, so that:

```
V_coordmap = mm2vxB.composewith(T_coordmap.composedwith(vx2mmA))
```

I have done a coregistration between two images, *img1* and *img2*. This has given me a voxel-to-voxel transformation and I want to store this transformation in such a way that I can use this transform to resample *img1* to *img2*. *Resampling use cases*

I have done a coregistration between two images, *img1* and *img2*. I may want this to give me a worldA-to-worldB transformation, where worldA is the world of voxel-to-world for *img1*, and worldB is the world of voxel-to-world of *img2*.

My *img1* has a voxel to world transformation. This transformation may (for example) have come from the scanner that acquired the image - so telling me how the voxel positions in *img1* correspond to physical coordinates in terms of the magnet isocenter and millimeters in terms of the primary gradient orientations (x, y and z). I have the same for *img2*. For example, I might choose to display this image resampled so each voxel is a 1mm cube.

Now I have these transformations: ST(*img1*-V2W), and ST(*img2*-V2W) (where ST is *scanner tranform* as above, and *V2W* is voxel to world).

I have now done a coregistration between *img1* and *img2* (somehow) - giving me, in addition to *img1* and *img2*, a transformation that registers *img1* and *img2*. Let's call this tranformation V2V(*img1*, *img2*), where V2V is voxel-to-voxel.

In actuality *img2* can be an array of images, such as series of fMRI images and I want to align all the *img2* series to *img1* and then take these voxel-to-voxel aligned images (the *img1* and *img2* array) and remap them to the world space (voxel-to-world). Since remapping is an interpolation operation I can generate errors in the resampled pixel values. If I do more than one resampling, error will accumulate. I want to do only a single resampling. To avoid the errors associated with resampling I will build a *composite transformation* that will chain the separate voxel-to-voxel and voxel-to-world transformations into a single transformation function (such as an affine matrix that is the result of multiplying the several affine matrices together). With this single *composite transformatio* I now resample *img1* and *img2* and put them into the world coordinate system from which I can make measurements.

10.2 Image model use cases

In which we lay out the various things that users and developers may want to do to images. See also *Resampling use cases*

I have a nifti image on disk and would like to load it and view it's header.

10.3 Resampling use cases

Use cases for image resampling. See also *Image model use cases*.

10.4 Batching use cases

Using the nipy framework for creating scripts to process whole datasets, for example movement correction, coregistration of functional to structural (intermodality), smoothing, statistics, inference.

CHAPTER

ELEVEN

DEVELOPER TOOLS

11.1 Tricked out emacs for python coding

Various ways to configure your emacs that you might find useful.

See emacs_python_mode for a good summary.

11.1.1 ReST mode

For editing ReST documents like this one. You may need a recent version of the rst.el file from the docutils site.

`rst` mode automates many important ReST tasks like building and updateing table-of-contents, and promoting or demoting section headings. Here is the basic `.emacs` configuration:

```
(require 'rst)
(setq auto-mode-alist
      (append '(("\\.txt$" . rst-mode)
                ("\\.rst$" . rst-mode)
                ("\\.rest$" . rst-mode)) auto-mode-alist))
```

Some helpful functions:

```
C-c TAB - rst-toc-insert

  Insert table of contents at point

C-c C-u - rst-toc-update

    Update the table of contents at point

C-c C-l rst-shift-region-left

    Shift region to the left

C-c C-r rst-shift-region-right

    Shift region to the right
```

On - for example - Ubuntu, the default `M-x rst-compile` command uses `rst2html.py` whereas the command installed is `rst2html`. Time for a symlink.

11.1.2 doctest mode

This useful mode for writing doctests (`doctest-mode.el`) came with my `python-mode` package in Ubuntu. Or see doctest-mode project page.

11.1.3 code checkers

Code checkers within emacs can be useful to check code for errors, unused variables, imports and so on. Alternatives are pychecker, pylint and pyflakes. Note that rope (below) also does some code checking. pylint and pyflakes work best with emacs flymake, which usually comes with emacs.

pychecker

This appears to be plumbed in with `python-mode`, just do `M-x py-pychecker-run`. If you try this, and pychecker is not installed, you will get an error. You can install it using your package manager (`pychecker` in Ubuntu) or from the pychecker webpage.

pylint

Install pylint. Ubuntu packages pylint as `pylint`. Put the flymake .emacs snippet in your `.emacs` file. You will see, in the emacs_python_mode page, that you will need to save this:

```
#!/usr/bin/env python

import re
import sys

from subprocess import *

p = Popen("pylint -f parseable -r n --disable-msg-cat=C,R %s" %
          sys.argv[1], shell = True, stdout = PIPE).stdout

for line in p.readlines():
    match = re.search("\\[([WE])(, (.+?))?\\]", line)
    if match:
        kind = match.group(1)
        func = match.group(3)

        if kind == "W":
           msg = "Warning"
        else:
           msg = "Error"

        if func:
            line = re.sub("\\[([WE])(, (.+?))?\\]",
                          "%s (%s):" % (msg, func), line)
        else:
            line = re.sub("\\[([WE])?\\]", "%s:" % msg, line)
    print line,

p.close()
```

as `epylint` somewhere on your system path, and test that `epylint somepyfile.py` works.

pyflakes

Install pyflakes. Maybe your package manager again? (`apt-get install pyflakes`). Install the flymake .emacs snippet in your `.emacs` file.

flymake .emacs snippet

Add this to your .emacs file:

```
;; code checking via flymake
;; set code checker here from "epylint", "pyflakes"
(setq pycodechecker "pyflakes")
(when (load "flymake" t)
  (defun flymake-pycodecheck-init ()
    (let* ((temp-file (flymake-init-create-temp-buffer-copy
                       'flymake-create-temp-inplace))
           (local-file (file-relative-name
                        temp-file
                        (file-name-directory buffer-file-name))))
      (list pycodechecker (list local-file))))
  (add-to-list 'flymake-allowed-file-name-masks
               '("\\.py\\'" flymake-pycodecheck-init)))
```

and set which of pylint ("epylint") or pyflakes ("pyflakes") you want to use.

You may also consider using the `flymake-cursor` functions, see the `pyflakes` section of the emacs_python_mode page for details.

11.1.4 ropemacs

rope is a python refactoring library, and ropemacs is an emacs interface to it, that uses pymacs. pymacs is an interface between emacs lisp and python that allows emacs to call into python and python to call back into emacs.

Install

- rope - by downloading from the link, and running `python setup.py install` in the usual way.
- pymacs - probably via your package manager - for example `apt-get install pymacs`
- ropemacs - download from link, `python setup.py install`

You may need to make sure your gnome etc sessions have the correct python path settings - for example settings in `.gnomerc` as well as the usual `.bashrc`.

Make sure you can *import ropemacs* from python (which should drop you into something lispey). Add these lines somewhere in your *.emacs* file:

```
(require 'pymacs)
(pymacs-load "ropemacs" "rope-")
```

and restart emacs. When you open a python file, you should have a `rope` menu. Note *C-c g* - the excellent *goto-definition* command.

11.1.5 Switching between modes

You may well find it useful to be able to switch fluidly between python mode, doctest mode, ReST mode and flymake mode (pylint). You can attach these modes to function keys in your `.emacs` file with something like:

```
(global-set-key [f8]      'flymake-mode)
(global-set-key [f9]      'python-mode)
(global-set-key [f10]      'doctest-mode)
(global-set-key [f11]      'rst-mode)
```

11.1.6 emacs code browser

Not really python specific, but a rather nice set of windows for browsing code directories, and code - see the ECB page. Again, your package manager may help you (`apt-get install ecb`).

11.2 Setting up virtualenv

Contents

11.2.1 Overview

virtualenv is a tool that allows you to install python packages in isolated environments. In this way you can have multiple versions of the same package without interference. I started using this to easily switch between multiple versions of numpy without having to constantly reinstall and update my symlinks. I also did this as a way to install software for Scipy2008, like the Enthought Tool Suite (ETS), in a way that would not effect my current development environment.

This tutorial is based heavily on a blog entry from Prabhu. I've extended his shell script to make switching between virtual environments a one-command operation. (Few others who should be credited for encouraging me to use virtualenv: Gael, Jarrod, Fernando)

11.2.2 Installing

Download and install the tarball for virtualenv:

```
tar xzf virtualenv-1.1.tar.gz
cd virtualenv-1.1
python setup.py install --prefix=$HOME/local
```

Note: I install in a local directory, your install location may differ.

11.2.3 Setup virtualenv

Setup a base virtualenv directory. I create this in a local directory, you can do this in a place of your choosing. All virtual environments will be installed as subdirectories in here.:

```
cd ~/local
mkdir -p virtualenv
```

11.2.4 Create a virtualenv

Create a virtual environment. Here I change into my virtualenv directory and create a virtual environment for my numpy-1.1.1 install:

```
cd virtualenv/
virtualenv numpy-1.1.1
```

11.2.5 Activate a virtualenv

Set the numpy-1.1.1 as the *active* virtual environment:

```
ln -s numpy-1.1.1/bin/activate .
```

We *enable* the numpy-1.1.1 virtual environment by sourcing it's activate script. This will prepend our *PATH* with the currently active virtual environment.:

```
# note: still in the ~/local/virtualenv directory
source activate
```

We can see our *PATH* with the numpy-1.1.1 virtual environment at the beginning. Also not the label of the virtual environment prepends our prompt.:

```
(numpy-1.1.1)cburns@~ 20:23:54 $ echo $PATH
/Users/cburns/local/virtualenv/numpy-1.1.1/bin:
/Library/Frameworks/Python.framework/Versions/Current/bin:
/Users/cburns/local/bin:
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin:/usr/local/git/bin
```

11.2.6 Install packages into a virtualenv

Then we install numpy-1.1.1 into the virtual environment. In order to install packages in the virtual environment, you need to use the *python* or *easy_install* from that virtualenv.:

```
~/local/virtualenv/numpy-1.1.1/bin/python setup.py install
```

At this point any package I install in this virtual environment will only be used when the environment is active.

11.2.7 Pragmatic virtualenv

There are a few more manual steps in the above process then I wanted, so I extended the shell script that Prabhu wrote to make this a simple one-command operation. One still needs to manually create each virtual environment, and install packages, but this script simplifies activating and deactivating them.

The *venv_switch.sh* script will:

- Activate the selected virtual environment. (Or issue an error if it doesn't exist.)
- Launch a new bash shell using the ~/.virtualenvrc file which sources the virtualenv/activate script.
- The activate script modifies the PATH and prepends the bash prompt with the virtualenv label.

venv_switch.sh:

```
#!/bin/sh
# venv_switch.sh
# switch between different virtual environments

# verify a virtualenv is passed in
if [ $# -ne 1 ]
then
    echo 'Usage: venv_switch venv-label'
    exit -1
fi

# verify the virtualenv exists
VENV_PATH=~/local/virtualenv/$1

# activate env script
ACTIVATE_ENV=~/local/virtualenv/activate

echo $VENV_PATH
if [ -e $VENV_PATH ]
then
    echo 'Switching to virtualenv' $VENV_PATH
    echo "Starting new bash shell.  Simply 'exit' to return to previous shell"
else
    echo 'Error: virtualenv' $VENV_PATH 'does not exist!'
    exit -1
fi

rm $ACTIVATE_ENV
ln -s ~/local/virtualenv/$1/bin/activate $ACTIVATE_ENV

# Launch new terminal
bash --rcfile ~/.virtualenvrc
```

Now to activate our numpy-1.1.1 virtual environment, we simply do:

```
venv_switch.sh numpy-1.1.1
```

To deactivate the virtual environment and go back to your original environment, just exit the bash shell:

```
exit
```

The rcfile used to source the activate script. I first source my .profile to setup my environment and custom prompt, then source the virtual environment. *.virtualenvrc*:

```
# rc file to initialize bash environment for virtualenv sessions

# first source the bash_profile
source ~/.bash_profile

# source the virtualenv
source ~/local/virtualenv/activate
```

11.2.8 Installing ETS 3.0.0

As another example, I installed ETS 3.0.0 for the Tutorial sessions at Scipy2008. (Note the prerequisites.)

Set up an ets-3.0.0 virtualenv:

```
cburns@virtualenv 15:23:50 $ pwd
/Users/cburns/local/virtualenv

cburns@virtualenv 15:23:50 $ virtualenv ets-3.0.0
New python executable in ets-3.0.0/bin/python
Installing setuptools.............done.

cburns@virtualenv 15:24:29 $ ls
activate      ets-3.0.0       numpy-1.1.1     numpy-1.2.0b2
```

Switch into my ets-3.0.0 virtualenv using the *venv_switch.sh* script:

```
cburns@~ 15:29:12 $ venv_switch.sh ets-3.0.0
/Users/cburns/local/virtualenv/ets-3.0.0
Switching to virtualenv /Users/cburns/local/virtualenv/ets-3.0.0
Starting new bash shell.  Simply 'exit' to return to previous shell
```

Install ETS using easy_install. Note we need to use the easy_install from our ets-3.0.0 virtual environment:

```
(ets-3.0.0)cburns@~ 15:31:41 $ which easy_install
/Users/cburns/local/virtualenv/ets-3.0.0/bin/easy_install

(ets-3.0.0)cburns@~ 15:31:48 $ easy_install ETS
```

11.3 SSH under Windows

To use ssh with bzr under windows you need to:

- Install Putty.
- Add the putty.exe directory to your path
- Add a environment variable BZR_SSH with value plink
- Generate a RSA ssh-2 key using puttygen.exe
- You need to start pageant and add you have registered on bzr.

CHAPTER

TWELVE

PYNIFTI IO

We are using pynifti for the underlying nifti file I/O support. Pynifti is built upon the nifticlibs so this will provide us will *full* nifti support. We are working closely with the author of pynifti, Michael Hanke, and pushing any bug fixes or feature improvements upstream to the git repository.

Developers should read through Michael's documentation on the pynifti site for some details on the project. The source is checked out from the git repository.

Using the command:

```
git clone http://git.debian.org/git/pkg-exppsy/pynifti.git
```

12.1 Git

Pynifti uses git for it's version control system. Git is very different from svn, developers should read some documentation on git before doing any work with the git repository.

The git website has several tutorials and full documentation. A good starting point may be the Git for SVN Users

Git has a unique mechanism for storing multiple branches on your machine. Instead of having separate file directories, git will store all branches in one directory and store *branch diffs* in an internal database. When you switch branches (`git checkout branch-name`), git will apply the branch diff to the directory, updating any files to match the new branch.

Git uses man pages for it's installed documentation. As with all man pages, these contain a lot of useful information, so you should know how to access them. All git commands can be called in two forms:

1. git add <filename>
2. git-add <filename>

The first form is the one you will probably use most and is what is often shown in the documentation. The second form, however, is what you need to access the man page. To see the man page on how to add a file to a git repository:

```
man git-add
```

To see a list of all git commands look at the main git man page:

```
man git
```

As with Bazaar, you should identify yourself to git so the repository keeps track of who made your edits:

```
git config --global user.name "Your Name Comes Here"
git config --global user.email you@yourdomain.example.com
```

To list your git configuration:

```
cburns@pynifti 13:32:31 $ git config -l
user.name=Christopher Burns
user.email=cburns[at]berkeley[dot]edu
color.diff=auto
color.status=auto
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=ssh://git.debian.org/git/pkg-exppsy/pynifti.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

We can also see the remote origin branch from which we have cloned.

12.2 Development Cycle

There are 3 development branches that the nipy developers need to interact with in the git repository:

- master - the main pynifti repository
- cb/master - Chris' pynifti developer repository
- cb/nipy - Chris' pynifti developer repository with nipy specific code

A nipy development path would look like this:

1. In the master branch, merge with the server to get any updates from other pynifti developers.
2. Checkout the cb/master branch and merge from the local master branch.
3. Make any code edits that should be pushed upstream in the cb/master branch. Michael cherry-picks changes into the master branch.
4. Checkout the cb/nipy branch and merge from the cb/master. The cb/nipy branch is used as the source for nipy.

12.2.1 Example:

I'm in my pynifti source directory:

```
cburns@pynifti 13:20:35 $ pwd
/Users/cburns/src/pynifti
```

Use the `git branch` command without arguments to see all of your local branches. Below we can see that I'm in my `cb/master` branch:

```
cburns@pynifti 13:20:39 $ git branch
* cb/master
  cb/nipy
  master
```

I want to switch to the `master` branch and update it with the server:

```
cburns@pynifti 13:26:08 $ git checkout master
Switched to branch "master"
cburns@pynifti 13:26:16 $ git branch
  cb/master
  cb/nipy
* master
```

Pull from the server to update our master branch:

```
cburns@pynifti 13:35:52 $ git pull
Password:
Already up-to-date.
```

Switch into `cb/master` and merge with the `master` branch. Remember, these are not separate directories, git *knows* about the other branch by name, so we do not specify a path, we specify a branch name.:

```
cburns@pynifti 13:36:18 $ git checkout cb/master
Switched to branch "cb/master"

cburns@pynifti 13:38:38 $ git merge master
Already up-to-date.
```

12.2.2 To update the nipy source:

1. Change to the pynifti directory in the nipy developer trunk:

   ```
   cd trunk-dev/neuroimaging/externals/pynifti/utils
   ```

2. Run the `update_source.py` script to update the source. This assumes a directory structure the pynifti sources are in the directory `$HOME/src/pynifti`. Run the script:

   ```
   python update_source.py
   ```

Part III

FAQ

CHAPTER

THIRTEEN

WHY ...

13.1 Why nipy?

We are writing NIPY because we hope that it will solve several problems in the field at the moment.

We are concentrating on FMRI analysis, so we'll put the case for that part of neuroimaging for now.

There are several good FMRI analysis packages already - for example *SPM*, *FSL* and *AFNI*. For each of these you can download the source code.

Like SPM, AFNI and FSL, we think source code is essential for understanding and development.

With these packages you can do many analyses. Some problems are that:

- The packages don't mix easily. You'll have to write your own scripts to mix between them; this is time-consuming and error-prone, because you will need good understanding of each package
- Because they don't mix, researchers usually don't try and search out the best algorithm for their task - instead they rely on the software that they are used to
- Each package has its own user community, so it's a little more difficult to share software and ideas
- The core development of each language belongs in a single lab.

Another, more general problem, is planning for the future. We need a platform that can be the basis for large scale shared development. For various reasons, it isn't obvious to us that any of these three is a good choice for common, shared development. In particular, we think that Python is the obvious choice for a large open-source software project. By comparison, matlab is not sufficiently general or well-designed as a programming language, and C / C++ are too hard and slow for scientific programmers to read or write. See why-python for this argument in more detail.

We started NIPY because we want to be able to:

- support an open collaborative development environment. To do this, we will have to make our code very easy to understand, modify and extend. If make our code available, but we are the only people who write or extend it, in practice, that is closed sofware.
- make the tools that allow developers to pick up basic building blocks for common tasks such as registration and statistics, and build new tools on top.
- write a scripting interface that allows you to mix in routines from the other packages that you like or that you think are better than the ones we have.
- design ways of interacting with the data and analysis stream that help you organize both. That way you can more easily keep track of your analyses. We also hope this will make analyses easier to run in parallel, and therefore much faster.

13.2 Why python?

The choice of programming language has many scientific and practical consequences. Matlab is an example of a high-level language. Languages are considered high level if they are able to express a large amount of functionality per line of code; other examples of high level languages are Python, Perl, Octave, R and IDL. In contrast, C is a low-level language. Low level languages can achieve higher execution speed, but at the cost of code that is considerably more difficult to read. C++ and Java occupy the middle ground sharing the advantages and the disadvantages of both levels.

Low level languages are a particularly ill-suited for exploratory scientific computing, because they present a high barrier to access by scientists that are not specialist programmers. Low-level code is difficult to read and write, which slows development ([Prechelt2000ECS], [boehm1981], [Walston1977MPM]) and makes it more difficult to understand the implementation of analysis algorithms. Ultimately this makes it less likely that scientists will use these languages for development, as their time for learning a new language or code base is at a premium. Low level languages do not usually offer an interactive command line, making data exploration much more rigid. Finally, applications written in low level languages tend to have more bugs, as bugs per line of code is approximately constant across many languages [brooks78].

In contrast, interpreted, high-level languages tend to have easy-to-read syntax and the native ability to interact with data structures and objects with a wide range of built-in functionality. High level code is designed to be closer to the level of the ideas we are trying to implement, so the developer spends more time thinking about what the code does rather than how to write it. This is particularly important as it is researchers and scientists who will serve as the main developers of scientific analysis software. The fast development time of high-level programs makes it much easier to test new ideas with prototypes. Their interactive nature allows researchers flexible ways to explore their data.

SPM is written in Matlab, which is a high-level language specialized for matrix algebra. Matlab code can be quick to develop and is relatively easy to read. However, Matlab is not suitable as a basis for a large-scale common development environment. The language is proprietary and the source code is not available, so researchers do not have access to core algorithms making bugs in the core very difficult to find and fix. Many scientific developers prefer to write code that can be freely used on any computer and avoid proprietary languages. Matlab has structural deficiencies for large projects: it lacks scalability and is poor at managing complex data structures needed for neuroimaging research. While it has the ability to integrate with other languages (e.g., C/C++ and FORTRAN) this feature is quite impoverished. Furthermore, its memory handling is weak and it lacks pointers - a major problem for dealing with the very large data structures that are often needed in neuroimaging. Matlab is also a poor choice for many applications such as system tasks, database programming, web interaction, and parallel computing. Finally, Matlab has weak GUI tools, which are crucial to researchers for productive interactions with their data.

CHAPTER

FOURTEEN

LICENSING

14.1 How do you spell licence?

If you are British you spell it differently from Americans, sometimes:

http://www.tiscali.co.uk/reference/dictionaries/english/data/d0082350.html

As usual the American spelling rule (always use *s*) was less painful and arbitrary, so I (MB) went for that.

14.2 Why did you choose BSD?

We have chosen BSD licensing, for compatibility with SciPy, and to increase input from developers in industry. Wherever possible we will keep packages that can have BSD licensing separate from packages needing a GPL license.

Our choices were between:

- *BSD*
- *GPL*

John Hunter made the argument for the BSD license in *Why we should be using BSD*, and we agree. Richard Stallman makes the case for the GPL here: http://www.gnu.org/licenses/why-not-lgpl.html

14.3 How does the BSD license affect our relationship to other projects?

The BSD license allows other projects with virtually any license, including GPL, to use our code. BSD makes it more likely that we will attract support from companies, including open-source software companies, such as Enthought and Kitware.

Any part of our code that uses (links to) GPL code, should be in a separable package.

Note that we do not have this problem with *LGPL*, which allows us to link without ourselves having a GPL.

14.4 What license does the NIH prefer?

The NIH asks that software written with NIH money can be commercialized. Quoting from: NIH NATIONAL CENTERS FOR BIOMEDICAL COMPUTING grant application document:

> A software dissemination plan must be included in the application. There is no prescribed single license for software produced in this project. However NIH does have goals for software dissemination, and reviewers will be instructed to evaluate the dissemination plan relative to these goals:
>
> 1. The software should be freely available to biomedical researchers and educators in the non-profit sector, such as institutions of education, research institutes, and government laboratories.
>
> 2. The terms of software availability should permit the commercialization of enhanced or customized versions of the software, or incorporation of the software or pieces of it into other software packages.

There is more discussion of licensing in this na-mic presentation. See also these links (from the presentation):

- http://www.rosenlaw.com/oslbook.htm
- http://www.opensource.org
- http://wiki.na-mic.org/Wiki/index.php/NAMIC_Wiki:Community_Licensing

So far this might suggest that the NIH would prefer at least a BSD-like license, but the NIH has supported several GPL'd projects in imaging, *AFNI* being the most obvious example.

CHAPTER

FIFTEEN

DOCUMENTATION FAQ

15.1 Installing graphviz on OSX

The easiest way I found to do this was using MacPorts, all other methods caused python exceptions when attempting to write out the pngs in the inheritance_diagram.py functions. Just do:

```
sudo port install graphviz
```

And make sure your macports directory (`/opt/local/bin`) is in your PATH.

15.2 Error writing output on OSX

If you are getting an error during the **writing output...** phase of the documentation build you may have a problem with your graphviz install. The error may look something like:

```
**writing output...** about api/generated/gen
  api/generated/neuroimaging
  api/generated/neuroimaging.algorithms.fwhm Format: "png" not
  recognized. Use one of: canon cmap cmapx cmapx_np dia dot eps fig
  hpgl imap imap_np ismap mif mp pcl pic plain plain-ext ps ps2 svg
  svgz tk vml vmlz vtx xdot

...

Exception occurred:

File "/Users/cburns/src/nipy-repo/trunk-dev/doc/sphinxext/
inheritance_diagram.py", line 238, in generate_dot
  (name, self._format_node_options(this_node_options)))

IOError: [Errno 32] Broken pipe
```

Try installing graphviz using MacPorts. See the *Installing graphviz on OSX* for instructions.

15.3 Sphinx and reST gotchas

15.3.1 Docstrings

Sphinx and reST can be very picky about whitespace. For example, in the docstring below the *Parameters* section will render correctly, where the *Returns* section will not. By correctly I mean Sphinx will insert a link to the CoordinateSystem class in place of the cross-reference *:class:‘CoordinateSystem‘*. The *Returns* section will be rendered exactly as shown below with the *:class:* identifier and the backticks around CoordinateSystem. This section fails because of the missing whitespace between `product_coord_system` and the colon `:`.

```
Parameters
----------
coord_systems : sequence of :class:‘CoordinateSystem‘

Returns
-------
product_coord_system: :class:‘CoordinateSystem‘
```

Part IV

FFF2: Fast FMRI Functions

CHAPTER

SIXTEEN

MASK-EXTRACTION UTILITIES

The module `fff2.utils.mask` contains utilities to extract brain masks from fMRI data:

`compute_mask`
`compute_mask_files`
`compute_mask_sessions`

The `compute_mask_files()` and `compute_mask_sessions()` functions work with Nifti files rather than numpy ndarrays. This is convenient to reduce memory pressure when working with long time series, as there is no need to store the whole series in memory.

CHAPTER

SEVENTEEN

EMPIRICAL NULL

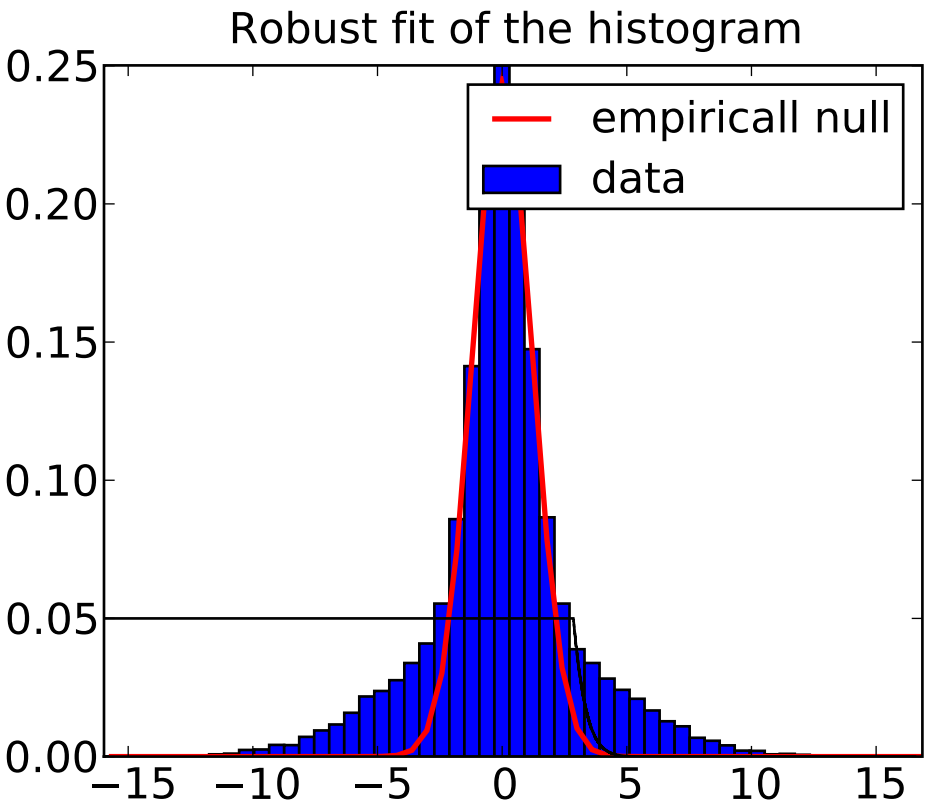
The `fff2.utils.emp_null` module contains a class that fits a gaussian model to the central part of an histogram, following Schwartzman et al, 2009. This is typically necessary to estimate a fdr when one is not certain that the data behaves as a standard normal under H_0.

The *ENN* class learns its null distribution on the data provided at initialisation. Two different methods can be used to set a threshold from the null distribution: the `ENN.threshold()` method returns the threshold for a given false discovery rate, and thus accounts for multiple comparisons with the given dataset; the `ENN.uncorrected_threshold()` returns the threshold for a given uncorrected p-value, and as such does not account for multiple comparisons.

17.1 Example

If we use the empirical normal null estimator on a two gaussian mixture distribution, with a central gaussian, and a wide one, it uses the central distribution as a null hypothesis, and returns the threshold followingr which the data can be claimed to belong to the wide gaussian:

```
import numpy as np
x = np.c_[np.random.normal(size=1e4),
          np.random.normal(scale=4, size=1e4)]

from nipy.neurospin.utils.emp_null import ENN
enn = ENN(x)
enn.threshold(verbose=True)
```



The threshold evaluated with the `ENN.threshold()` method is around 2.8 (using the default p-value of 0.05). The `ENN.uncorrected_threshold()` return, for the same p-value, a threshold of 1.9. It is necessary to use a higher p-value with uncorrected comparisons.

17.2 Class documentation

Reference: Schwartzmann et al., NeuroImage 44 (2009) 71–82

CHAPTER

EIGHTEEN

AUTOMATIC PLOTTING OF ACTIVATION MAPS

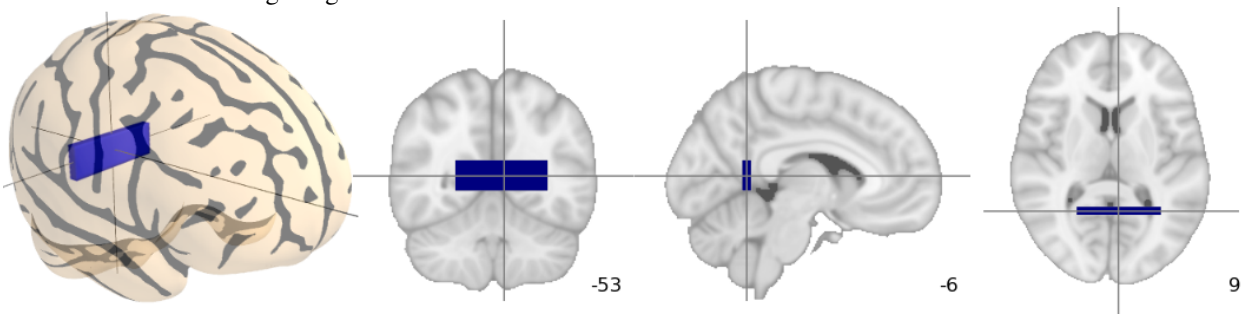
The module `fff2.viz.activation_maps` provides functions to plot visualization of activation maps in a non-interactive way.

2D cuts of an activation map can be plotted and superimposed on an anatomical map using matplotlib. In addition, Mayavi2 can be used to plot 3D maps, using volumetric rendering. Some emphasis is made on automatic choice of default parameters, such as cut coordinates, to give a sensible view of a map in a purely automatic way, for instance to save a summary of the output of a calculation.

18.1 An example

```
from fff2.viz.activation_maps import plot_map, mni_sform, \
        coord_transform

# First, create a fake activation map: a 3D image in MNI space with
# a large rectangle of activation around Brodmann Area 26
import numpy as np
mni_sform_inv = np.linalg.inv(mni_sform)
map = np.zeros((182, 218, 182))
x, y, z = -6, -53, 9
x_map, y_map, z_map = coord_transform(x, y, z, mni_sform_inv)
map[x_map-30:x_map+30, y_map-3:y_map+3, z_map-10:z_map+10] = 1

# And now, visualize it:
plot_map(map, mni_sform, cut_coords=(x, y, z), vmin=0.5)
```

This creates the following image:



The same plot can be obtained fully automaticaly, by using `auto_plot_map()` to find the activation threshold *vmin* and the cut coordinnates:

```
from fff2.viz.activation_maps import auto_plot_map
auto_plot_map(map, mni_sform)
```

In this simple example, the code will easily detect the bar as activation and position the cut at the center of the bar.

18.2 fff2.viz.activation_maps functions

coord_transform	
find_activation	
find_cut_coords	
plot_map_2d	
plot_map_3d	
auto_plot_map	
plot_niftifile	

18.3 The plot_activation script

In addition to the above functions, callable from Python, there is also script **plot_activation** that can be used to easily render previews from Nifti files. It can also optionally output an html file, to summarize many maps on one screen.

In its simplest use, it is called like this:

```
plot_activation file1.nii file2.nii [...]
```

The **plot_activation** script has several many options:

-h, –help show the help message and exit

-o *DIR*, –outdir=DIR write all output to DIR

-f *FILE*, –htmlfile=FILE write report to a html file FILE, useful when visualizing multiple files

-a *FILE*, –anat=FILE use the given Nifti file as an anatomy

-M *FILE*, –mask=FILE use the given Nifti file as a mask

-d, –no3d don't try to do a 3D view

-s, –sign force activation sign to be positive

-c *CUT_COORDS*, –cut-coords=CUT_COORDS Talairach coordinates of the 2D cuts

-m *VMIN*, –vmin=VMIN Minimum value for the activation, used for thresholding

CHAPTER

NINETEEN

GENERATING SIMULATED ACTIVATION MAPS

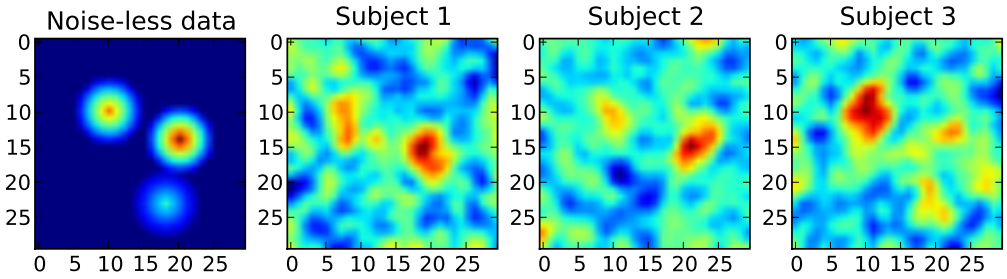
The module `fff2.utils.simul_2d_multisubject_fmri_dataset` contains a function to create simulated 2D activation maps: `make_surrogate_array()`. The function can position various activations and add noise, both as background noise and jitter in the activation positions and amplitude.

This function is useful to test methods.

19.1 Example

```
import numpy as np
import pylab as pl

from nipy.neurospin.utils.simul_2d_multisubject_fmri_dataset import \
     make_surrogate_array

pos = np.array([[10, 10],
                [14, 20],
                [23, 18]])
ampli = np.array([4, 5, 2])

# First generate some noiseless data
noiseless_data = make_surrogate_array(nbsubj=1, noise_level=0, spatial_jitter=0,
                                      signal_jitter=0, pos=pos, ampli=ampli)

pl.figure(figsize=(10, 3))
pl.subplot(1, 4, 1)
pl.imshow(noiseless_data[0])
pl.title('Noise-less data')

# Second, generate some group data, with default noise parameters
group_data = make_surrogate_array(nbsubj=3, pos=pos, ampli=ampli)

pl.subplot(1, 4, 2)
pl.imshow(group_data[0])
pl.title('Subject 1')
pl.subplot(1, 4, 3)
pl.title('Subject 2')
pl.imshow(group_data[1])
pl.subplot(1, 4, 4)
```

```
pl.title('Subject 3')
pl.imshow(group_data[2])
```



19.2 Function documentation

Part V

API

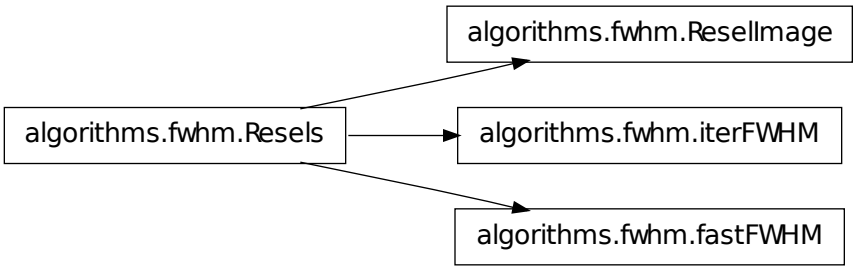
CHAPTER

TWENTY

ALGORITHMS.FWHM

20.1 Module: algorithms.fwhm

Inheritance diagram for nipy.algorithms.fwhm:



This module provides classes and definitions for using full width at half maximum (FWHM) to be used in conjunction with Gaussian Random Field Theory to determine resolution elements (resels).

A resolution element (resel) is defined as a block of pixels of the same size as the FWHM of the smoothed image.

There are two methods implemented to estimate (3d, or volumewise) FWHM based on a 4d Image:

> fastFHWM: used if the entire 4d Image is available iterFWHM: used when 4d Image is being filled in by slices of residuals

20.2 Classes

20.2.1 ReselImage

class ReselImage(*resels=None, fwhm=None, **keywords*)

Bases: nipy.algorithms.fwhm.Resels

__init__(*resels=None, fwhm=None, **keywords*)

Parameters **resels** [*core.api.Image*] Image of resel per voxel values.

fwhm [*core.api.Image*] Image of FWHM values.

keywords [dict] Passed as keywords arguments to *core.api.Image*

20.2.2 Resels

class Resels(*coordmap, normalized=False, fwhm=None, resels=None, mask=None, clobber=False, D=3*)
Bases: object

The Resels class.

__init__(*coordmap, normalized=False, fwhm=None, resels=None, mask=None, clobber=False, D=3*)

Parameters **coordmap** [CoordinateMap] CoordinateMap over which fwhm and resels are to be estimated. Used in fwhm/resel conversion.

fwhm [Image] Optional Image of FWHM. Used to convert FWHM Image to resels if FWHM is not being estimated.

resels [Image] Optional Image of resels. Used to compute resels within a mask, for instance, if FWHM has already been estimated.

mask [Image] Mask over which to integrate resels.

clobber [bool] Clobber output FWHM and resel images?

D [int] Can be 2 or 3, the dimension of the final volume.

fwhm2resel(*fwhm*)

Parameters **fwhm** [float] Convert an FWHM value to an equivalent resels per voxel based on step sizes in self.coordmap.

Returns resels

integrate(*mask=None*)

Parameters **mask** [Image] Optional mask over which to integrate (add) resels.

Returns (total_resels, FWHM, nvoxel)

total_resels: the resels contained in the mask FWHM: an estimate of FWHM based on the average resel per voxel nvoxel: the number of voxels in the mask

resel2fwhm(*resels*)

Parameters **resels** [float] Convert a resel value to an equivalent isotropic FWHM based on step sizes in self.coordmap.

Returns FWHM

20.2.3 fastFWHM

class fastFWHM(*resid, **keywords*)
Bases: nipy.algorithms.fwhm.Resels

__init__(*resid, **keywords*)
Given a 4d image of residuals, i.e. not one filled in step by step by an iterator, estimate FWHM and resels.

Parameters **resid** [array] Image of residuals used to estimate FWHM and resels per voxel.

Returns None

20.2.4 iterFWHM

class iterFWHM(*coordmap, normalized=False, df_resid=5.0, mask=None, **keywords*)
Bases: nipy.algorithms.fwhm.Resels

Estimate FWHM on an image of residuals sequentially. This is handy when, say, residuals from a linear model are written out slice-by-slice.

Resulting FWHM is clipped at self.FWHMmax, which defaults to 50.

__init__(*coordmap, normalized=False, df_resid=5.0, mask=None, **keywords*)
Setup a FWHM estimator.

Parameters **coordmap** [CoordinateMap] CoordinateMap over which fwhm and resels are to be estimated. Used in fwhm/resel conversion.

normalized [bool] Are residuals normalized to have length 1? If False, residuals are normalized before estimating FWHM.

df_resid [float] How many degrees of freedom are there in the residuals? Must be greater than self.D + 1.

mask [Image] Optional mask over which to integrate (add) resels.

keywords [dict] Passed as keyword parameters to *Resels.__init__*

normalize(*resid*)
Normalize residuals subtracting mean, and fixing length to 1.

Parameters resid : Array of residuals.

Returns Normalized residuals.

output()

Returns None

set_next(*resid*)
Pass a slice of residuals into slicewise estimate of FWHM.

Parameters **resid** [array] slice of residuals

Returns None

CHAPTER

TWENTYONE

ALGORITHMS.INTERPOLATION

21.1 Module: algorithms.interpolation

Inheritance diagram for nipy.algorithms.interpolation:

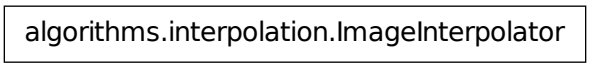


Image interpolators using ndimage.

21.2 ImageInterpolator

class ImageInterpolator(*image, order=3*)

Bases: object

A class that enables interpolation of an Image instance at arbitrary points in the Image's world space (image.coordmap.output_coords).

The resampling is done with scipy.ndimage.

__init__(*image, order=3*)

Parameters **image** [Image] Image to be interpolated

order [int] order of spline interpolation as used in scipy.ndimage

evaluate(*points*)

Parameters points : values in self.image.coordmap.output_coords

Returns **V: ndarray** interpolator of self.image evaluated at points

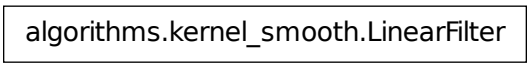
CHAPTER

TWENTYTWO

ALGORITHMS.KERNEL_SMOOTH

22.1 Module: algorithms.kernel_smooth

Inheritance diagram for nipy.algorithms.kernel_smooth:



TODO

22.2 Class

22.3 LinearFilter

class LinearFilter(*coordmap, shape, fwhm=6.0, scale=1.0, location=0.0, cov=None*)

Bases: object

A class to implement some FFT smoothers for Image objects. By default, this does a Gaussian kernel smooth. More choices would be better!

__init__(*coordmap, shape, fwhm=6.0, scale=1.0, location=0.0, cov=None*)

Parameters **coordmap** [TODO] TODO

fwhm [float] TODO

scale [float] TODO

location [float] TODO

smooth(*inimage, clean=False, is_fft=False*)

Parameters **inimage** [*core.api.Image*] The image to be smoothed

clean [bool] Should we call nan_to_num on the data before smoothing?

is_fft [bool] Has the data already been fft'd?

Returns *Image*

22.4 Functions

fwhm2sigma(*fwhm*)
Convert a FWHM value to sigma in a Gaussian kernel.

Parameters **fwhm** [float] TODO

Returns float

sigma2fwhm(*sigma*)
Convert a sigma in a Gaussian kernel to a FWHM value.

Parameters sigma : float

Returns float

CHAPTER

TWENTYTHREE

ALGORITHMS.RESAMPLE

23.1 Module: `algorithms.resample`

Some simple examples and utility functions for resampling.

`resample`(*image, target, mapping, shape, order=3*)

Resample an image to a target CoordinateMap with a "world-to-world" mapping and spline interpolation of a given order.

Here, "world-to-world" refers to the fact that mapping should be a callable that takes a physical coordinate in "target" and gives a physical coordinate in "image".

Parameters **image** : Image instance that is to be resampled

target :target CoordinateMap for output image :

mapping : transformation from target.output_coords

to image.coordmap.output_coords, i.e. 'world-to-world mapping' Can be specified in three ways: a callable, a tuple (A, b) representing the mapping y=dot(A,x)+b or a representation of this in homogeneous coordinates.

shape : shape of output array, in target.input_coords

order : what order of interpolation to use in *scipy.ndimage*

Returns **output** : Image instance with interpolated data and output.coordmap == target

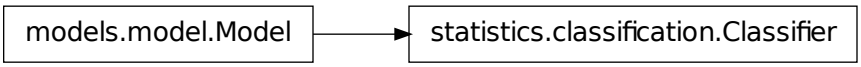
CHAPTER

TWENTYFOUR

ALGORITHMS.STATISTICS.CLASSIFICATION

24.1 Module: algorithms.statistics.classification

Inheritance diagram for nipy.algorithms.statistics.classification:



TODO

24.2 Classifier

class Classifier()

Bases: nipy.fixes.scipy.stats.models.model.Model

TODO

__init__()

learn(***keywords*)

Parameters **keywords** [dict] TODO

Returns None

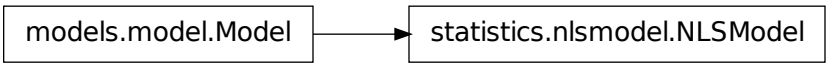
CHAPTER

TWENTYFIVE

ALGORITHMS.STATISTICS.NLSMODEL

25.1 Module: algorithms.statistics.nlsmodel

Inheritance diagram for nipy.algorithms.statistics.nlsmodel:



TODO

25.2 NLSModel

class NLSModel(*Y, design, f, grad, theta, niter=10*)

Bases: nipy.fixes.scipy.stats.models.model.Model

Class representing a simple nonlinear least squares model.

__init__(*Y, design, f, grad, theta, niter=10*)

Parameters **Y** [TODO] the data in the NLS model

design [TODO] the deisng matrix, X

f [TODO] the map between the linear parameters (in the design matrix) and the nonlinear parameters (theta)

grad [TODO] the gradient of f, this should be a function of an nxp design matrix X and qx1 vector theta that returns an nxq matrix df_i/dtheta_j where

f_i(theta) = f(X[i], theta)

is the nonlinear response function for the i-th instance in the model.

SSE()

Sum of squares error.

:Returns; TODO

getZ()

Returns None

getomega()

Returns None

next()

Returns None

predict(*design=None*)

Parameters **design** [TODO] TODO

Returns TODO

CHAPTER

TWENTYSIX

ALGORITHMS.STATISTICS.ONESAMPLE

26.1 Module: algorithms.statistics.onesample

TODO

26.2 Functions

estimate_mean(*Y, sd*)

Estimate the mean of a sample given information about the standard deviations of each entry.

Parameters **Y** : np.ndarray

Data for which mean is to be estimated. Should have shape (nsubj, nvox).

sd : np.ndarray

Standard deviation (subject specific) of the data for which the mean is to be estimated. Should have shape (nsubj, nvox).

Returns **value** : dict

This dictionary has keys ['mu', 'scale', 't', 'resid', 'sd']

estimate_varatio(*Y, sd, df=None, niter=10*)

In a one-sample random effects problem, estimate the ratio between the fixed effects variance and the random effects variance.

Parameters **Y** : np.ndarray

Data for which mean is to be estimated. Should have shape (nsubj, nvox).

sd : np.ndarray

Standard deviation (subject specific) of the data for which the mean is to be estimated. Should have shape (nsubj, nvox).

df : [int]

If supplied, these are used as weights when deriving the fixed effects variance. Should have length [nsubj].

niter : int

Number of EM iterations to perform.

Returns **value** : dict

This dictionary has keys ['fix', 'ratio', 'random'], where 'fix' is the fixed effects variance implied by the input parameter 'sd'; 'random' is the random effects variance and 'ratio' is the estimated ratio of variances: 'random'/'fixed'.

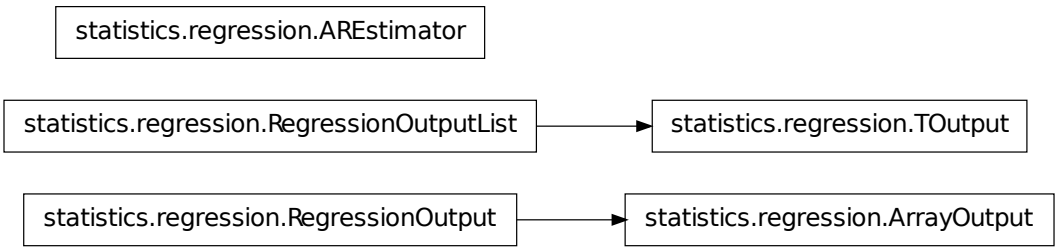
CHAPTER

TWENTYSEVEN

ALGORITHMS.STATISTICS.REGRESSION

27.1 Module: algorithms.statistics.regression

Inheritance diagram for nipy.algorithms.statistics.regression:



This module provides various convenience functions for extracting statistics from regression analysis techniques to model the relationship between the dependent and independent variables.

As well as a convenience class to output the result, RegressionOutput

27.2 Classes

27.2.1 AREstimator

class AREstimator (*model, p=1*)

A class that whose instances can estimate AR(p) coefficients from residuals

__init__ (*model, p=1*)

Parameters ***coordmap*** [TODO] TODO

model [TODO] A scipy.stats.models.regression.OLSmodel instance

p [int] Order of AR(p) noise

27.2.2 ArrayOutput

class ArrayOutput(*img, fn*)
Bases: nipy.algorithms.statistics.regression.RegressionOutput

Output an array from a GLM pass through data.

By default, the function called is output_resid, so residuals are output.

__init__(*img, fn*)

27.2.3 RegressionOutput

class RegressionOutput(*img, fn, output_shape=None*)
A class to output things in GLM passes through arrays of data.

__init__(*img, fn, output_shape=None*)

Parameters *img* : the output Image *fn* : a function that is applied to a scipy.stats.models.model.LikelihoodModelResults instance

27.2.4 RegressionOutputList

class RegressionOutputList(*imgs, fn*)
A class to output more than one thing from a GLM pass through arrays of data.

__init__(*imgs, fn*)

Parameters *imgs* : the list of output images *fn* : a function that is applied to a scipy.stats.models.model.LikelihoodModelResults instance

27.2.5 TOutput

class TOutput(*contrast, effect=None, sd=None, t=None*)
Bases: nipy.algorithms.statistics.regression.RegressionOutputList

Output contrast related to a T contrast from a GLM pass through data.

__init__(*contrast, effect=None, sd=None, t=None*)

27.3 Functions

output_AR1(*results*)
Compute the usual AR(1) parameter on the residuals from a regression.

output_F(*results, contrast*)
This convenience function outputs the results of an Fcontrast from a regression

output_T(*contrast, results, effect=None, sd=None, t=None*)
This convenience function outputs the results of a Tcontrast from a regression

output_resid(*results*)
This convenience function outputs the residuals from a regression

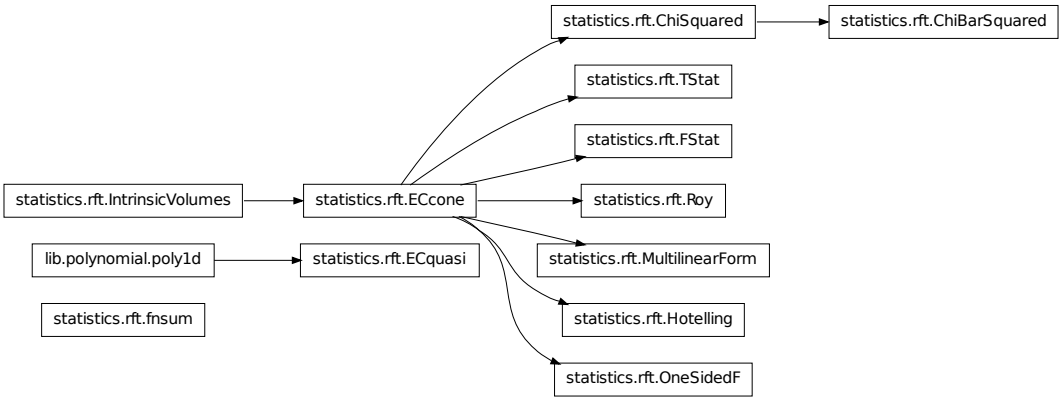
CHAPTER

TWENTYEIGHT

ALGORITHMS.STATISTICS.RFT

28.1 Module: algorithms.statistics.rft

Inheritance diagram for nipy.algorithms.statistics.rft:



28.2 Classes

28.2.1 ChiBarSquared

class ChiBarSquared(*dfn=1, search=, [1]*)

Bases: nipy.algorithms.statistics.rft.ChiSquared

__init__(*dfn=1, search=, [1]*)

28.2.2 ChiSquared

class ChiSquared(*dfn, dfd=inf, search=, [1]*)

Bases: nipy.algorithms.statistics.rft.ECcone

EC densities for a Chi-Squared(n) random field.

__init__(*dfn, dfd=inf, search=, [1]*)

28.2.3 ECcone

class ECcone(*mu=, [1], dfd=inf, search=, [1], product=, [1]*)
Bases: nipy.algorithms.statistics.rft.IntrinsicVolumes

A class that takes the intrinsic volumes of a set and gives the EC approximation to the supremum distribution of a unit variance Gaussian process with these intrinsic volumes. This is the basic building block of all of the EC densities.

If product is not None, then this product (an instance of IntrinsicVolumes) will effectively be prepended to the search region in any call, but it will also affect the (quasi-)polynomial part of the EC density. For instance, Hotelling's T^2 random field has a sphere as product, as does Roy's maximum root.

__init__(*mu=, [1], dfd=inf, search=, [1], product=, [1]*)

density(*x, dim*)
The EC density in dimension dim.

integ(*m=None, k=None*)

pvalue(*x, search=None*)

quasi(*dim*)
(Quasi-)polynomial parts of the EC density in dimension dim ignoring a factor of (2pi)^{-(dim+1)/2} in front.

28.2.4 ECquasi

class ECquasi(*c_or_r, r=0, exponent=None, m=None*)
Bases: numpy.lib.polynomial.poly1d

A subclass of poly1d consisting of polynomials with a premultiplier of the form

(1 + x^2/m)^-exponent

where m is a non-negative float (possibly infinity, in which case the function is a polynomial) and exponent is a non-negative multiple of 1/2.

These arise often in the EC densities.

__init__(*c_or_r, r=0, exponent=None, m=None*)

change_exponent(*_pow*)
Multiply top and bottom by an integer multiple of the self.denom_poly.

compatible(*other*)
Check whether the degrees of freedom of two instances are equal so that they can be multiplied together.

denom_poly()
This is the base of the premultiplier: (1+x^2/m).

deriv(*m=1*)
Evaluate derivative of ECquasi.

28.2.5 FStat

class FStat (*dfn, dfd=inf, search=, [1]*)
Bases: nipy.algorithms.statistics.rft.ECcone

EC densities for a F random field.

__init__ (*dfn, dfd=inf, search=, [1]*)

28.2.6 Hotelling

class Hotelling (*dfd=inf, k=1, search=, [1]*)
Bases: nipy.algorithms.statistics.rft.ECcone

Hotelling's T^2: maximize an F_{1,dfd}=T_dfd^2 statistic over a sphere of dimension k.

__init__ (*dfd=inf, k=1, search=, [1]*)

28.2.7 IntrinsicVolumes

class IntrinsicVolumes (*mu=, [1]*)
A simple class that exists only to compute the intrinsic volumes of products of sets (that themselves have intrinsic volumes, of course).

__init__ (*mu=, [1]*)

28.2.8 MultilinearForm

class MultilinearForm (**dims, **keywords*)
Bases: nipy.algorithms.statistics.rft.ECcone

Maximize a multivariate Gaussian form, maximized over spheres of dimension dims. See

Kuri, S. & Takemura, A. (2001). 'Tail probabilities of the maxima of multilinear forms and their applications.' Ann, Statist. 29(2): 328-371.

__init__ (**dims, **keywords*)

28.2.9 OneSidedF

class OneSidedF (*dfn, dfd=inf, search=, [1]*)
Bases: nipy.algorithms.statistics.rft.ECcone

EC densities for one-sided F statistic in

Worsley, K.J. & Taylor, J.E. (2005). 'Detecting fMRI activation allowing for unknown latency of the hemodynamic response.' Neuroimage, 29,649-654.

__init__ (*dfn, dfd=inf, search=, [1]*)

28.2.10 Roy

class Roy (*dfn=1, dfd=inf, k=1, search=, [1]*)
Bases: nipy.algorithms.statistics.rft.ECcone

Roy's maximum root: maximize an F_{dfd,dfn} statistic over a sphere of dimension k.

__init__ (*dfn=1, dfd=inf, k=1, search=, [1]*)

28.2.11 TStat

class TStat (*dfd=inf, search=, [1]*)

Bases: nipy.algorithms.statistics.rft.ECcone

EC densities for a t random field.

__init__ (*dfd=inf, search=, [1]*)

28.2.12 fnsum

class fnsum (**items*)

__init__ (**items*)

28.3 Functions

Q (*dim, dfd=inf*)

If dfd == inf (the default), then Q(dim) is the (dim-1)-st Hermite polynomial

H_j(x) = (-1)^j * e^{x^2/2} * (d^j/dx^j e^{-x^2/2})

If dfd != inf, then it is the polynomial Q defined in

Worsley, K.J. (1994). 'Local maxima and the expected Euler characteristic of excursion sets of chi^2, F and t fields.' Advances in Applied Probability, 26:13-42.

ball_search (*n, r=1*)

A ball-shaped search region of radius r.

binomial (*n, j*)

Binomial coefficient:

mu_ball (*n, j, r=1*)

Return mu_j(B_n(r)), the j-th Lipschitz Killing curvature of the ball of radius r in R^n.

mu_sphere (*n, j, r=1*)

Return mu_j(S_r(R^n)), the j-th Lipschitz Killing curvature of the sphere of radius r in R^n.

From Chapter 6 of

Adler & Taylor, 'Random Fields and Geometry'. 2006.

scale_space (*region, interval, kappa=1.0*)

Work out intrinsic volumes of region x interval in the scale space model. See

Siegmund, D.O and Worsley, K.J. (1995). 'Testing for a signal with unknown location and scale in a stationary Gaussian random field.' Annals of Statistics, 23:608-639.

and

Taylor, J.E. & Worsley, K.J. (2005). 'Random fields of multivariate test statistics, with applications to shape analysis and fMRI.'

(available on http://www.math.mcgill.ca/keith

spherical_search(*n, r=1*)
A spherical search region of radius r.

volume2ball(*vol, d=3*)
Approximate intrinsic volumes of a set with a given volume by those of a ball with a given dimension and equal volume.

CHAPTER

TWENTYNINE

ALGORITHMS.STATISTICS.UTILS

29.1 Module: algorithms.statistics.utils

29.2 Functions

combinations(*iterable, r*)

complex(*maximal=, [(0, 3, 2, 7), (0, 6, 2, 7), (0, 7, 5, 4), (0, 7, 5, 1), (0, 7, 4, 6), (0, 3, 1, 7)], vertices=None*)
Take a list of maximal simplices (by default a triangulation of a cube into 6 tetrahedra) and computes all faces, edges, vertices.

If vertices is not None, then the vertices in 'maximal' are replaced with these vertices, by index.

cube_with_strides_center(*center=, [0, 0, 0], strides=(4, 2, 1)*)
Cube in an array of voxels with a given center and strides.

This triangulates a cube with vertices [center[i] + 1].

The dimension of the cube is determined by len(center) which should agree with len(center).

The allowable dimensions are [1,2,3].

decompose2d(*shape, dim=3*)
Return all (dim-1)-dimensional simplices in a triangulation of a square of a given shape. The vertices in the triangulation are indices in a 'flattened' array of the specified shape.

decompose3d(*shape, dim=4*)
Return all (dim-1)-dimensional simplices in a triangulation of a cube of a given shape. The vertices in the triangulation are indices in a 'flattened' array of the specified shape.

join_complexes(**complexes*)
Join a sequence of simplicial complexes. Returns the union of all the particular faces.

test_EC2(*shape*)

test_EC3(*shape*)

CHAPTER

THIRTY

CORE.IMAGE.GENERATORS

30.1 Module: core.image.generators

This module defines a few common generators that may be useful for Image instances.

They are defined on ndarray, so they do not depend on Image.

data_generator: return (item, data[item]) tuples from an iterable object slice_generator: return slices through an ndarray, possibly over many indices f_generator: return a generator that applies a function to the output of another generator

The above three generators return 2-tuples.

write_data: write the output of a generator to an ndarray parcels: return binary array of the unique components of data

30.2 Functions

data_generator (*data, iterable=None*)

Return a generator for

[(i, data[i]) for i in iterable]

If iterables is None, it defaults to range(data.shape[0])

```
>>> a = asarray([[True,False],[False,True]])
>>> b = asarray([[False,False],[True,False]])


>>> for i, d in data_generator(asarray([[1,2],[3,4]]), [a,b]):
...     print d
...
[1 4]
[3]
>>>
```

f_generator (*f, iterable*)

Return a generator for

[(i, f(x)) for i, x in iterable]

```
>>> for i, d in f_generator(lambda x: x**2, data_generator([[1,2],[3,4]])):
...     print i, d
...
0 [1 4]
```

```
1 [ 9 16]
>>>
```

matrix_generator(*img*)
From a generator of items (i, r), return (i, rp) where rp is a 2d array with rp.shape = (r.shape[0], prod(r.shape[1:]))

parcels(*data, labels=None, exclude=, []*)
Take data and return a generator for

[numpy.equal(data, label) for label in labels]

If labels is None, labels = numpy.unique(data). Each label in labels can be a sequence, in which case the value returned for that label union [numpy.equal(data, l) for l in label]

```
>>> for p in parcels([[1,1],[2,1]]):
...     print p
...
[[ True  True]
 [False  True]]
[[False False]
 [ True False]]
>>>
>>> for p in parcels([[1,1],[2,3]], labels=[2,3]):
...     print p
...
[[False False]
 [ True False]]
[[False False]
 [False  True]]
>>>
>>> for p in parcels([[1,1],[2,3]], labels=[(2,3),2]):
...     print p
...
[[False False]
 [ True  True]]
[[False False]
 [ True False]]
>>>
```

shape_generator(*img, shape*)
From a generator of items (i, r), return (i, r.reshape(shape))

slice_generator(*data, axis=0*)
Create a generator for iterating through slices of data along a given axis.

```
>>> for i,d in slice_generator([[1,2],[3,4]]):
...     print i, d
...
(0,) [1 2]
(1,) [3 4]
>>> for i,d in slice_generator([[1,2],[3,4]], axis=1):
...     print i, d
...
(slice(None, None, None), 0) [1 3]
(slice(None, None, None), 1) [2 4]
>>>
```

slice_parcels(*data, labels=None, axis=0*)
A generator for slicing through parcels and slices of data...

hmmm... a better description is needed

```
>>> from numpy import *
>>> x=array([[0,0,0,1],[0,1,0,1],[2,2,0,1]])
>>> for a in slice_parcels(x):
...     print a, x[a]
...
((0,), array([ True,  True,  True, False], dtype=bool)) [0 0 0]
((0,), array([False, False, False,  True], dtype=bool)) [1]
((1,), array([ True, False,  True, False], dtype=bool)) [0 0]
((1,), array([False,  True, False,  True], dtype=bool)) [1 1]
((2,), array([False, False,  True, False], dtype=bool)) [0]
((2,), array([False, False, False,  True], dtype=bool)) [1]
((2,), array([ True,  True, False, False], dtype=bool)) [2 2]


>>> for a in slice_parcels(x, axis=1):
...     b, c = a
...     print a, x[b][c]
...
((slice(None, None, None), 0), array([ True,  True, False], dtype=bool)) [0 0]
((slice(None, None, None), 0), array([False, False,  True], dtype=bool)) [2]
((slice(None, None, None), 1), array([ True, False, False], dtype=bool)) [0]
((slice(None, None, None), 1), array([False,  True, False], dtype=bool)) [1]
((slice(None, None, None), 1), array([False, False,  True], dtype=bool)) [2]
((slice(None, None, None), 2), array([ True,  True,  True], dtype=bool)) [0 0 0]
((slice(None, None, None), 3), array([ True,  True,  True], dtype=bool)) [1 1 1]
```

write_data(*output, iterable*)

Write some data to output. Iterable should return 2-tuples of the form index, data such that

output[index] = data

makes sense.

```
>>> import numpy as np
>>> a=np.zeros((2,2))
>>> write_data(a, data_generator(asarray([[1,2],[3,4]])))
>>> a
array([[ 1.,  2.],
       [ 3.,  4.]])
```

CHAPTER

THIRTYONE

CORE.IMAGE.IMAGE

31.1 Module: core.image.image

Inheritance diagram for nipy.core.image.image:

This module defines the Image class, as well as two functions that create Image instances.

fromarray : create an Image instance from an ndarray

merge_images : create an Image by merging a sequence of Image instance

31.2 Class

31.3 Image

class Image(*data, coordmap*)

Bases: object

The *Image* class provides the core object type used in nipy. An *Image* represents a volumetric brain image and provides means for manipulating the image data. Most functions in the image module operate on *Image* objects.

Notes

Images should be created through the module functions load and fromarray.

Examples

```
>>> from nipy.core.image import image
>>> from nipy.testing import anatfile
>>> from nipy.io.api import load_image
>>> img = load_image(anatfile)
```

```
>>> import numpy as np
>>> img = image.fromarray(np.zeros((21, 64, 64), dtype='int16'),
...                       'kji', 'zxy')
```

__init__(*data, coordmap*)

Create an *Image* object from array and `CoordinateMap` object.

Images should be created through the module functions load and fromarray.

Parameters **data** : A numpy.ndarray

coordmap : A *CoordinateMap* Object

See Also:

load load *Image* from a file

save save *Image* to a file

fromarray create an *Image* from a numpy array

affine

Affine transformation is one exists

coordmap

Coordinate mapping from input coords to output coords

header

The file header dictionary for this image. In order to update the header, you must first make a copy of the header, set the values you wish to change, then set the image header to the updated header.

ndim

Number of data dimensions

shape

Shape of data array

31.4 Functions

fromarray(*data, innames, outnames, coordmap=None*)

Create an image from a numpy array.

Parameters **data** : numpy array

A numpy array of three dimensions.

names : a list of axis names

coordmap : A *CoordinateMap*

If not specified, an identity coordinate map is created.

Returns **image** : An *Image* object

See Also:

load function for loading images

save function for saving images

merge_images(*images, cls=<class 'nipy.core.image.image.Image'>, clobber=False, axis='merge'*)

Create a new file based image by combining a series of images together. The resulting CoordinateMap are essentially copies of images[0].coordmap

Parameters **images** : [*Image*]

The list of images to be merged

cls : `class`

The class of image to create

clobber : `bool`

Overwrite the file if it already exists

axis : `string`

Name of the concatenated axis.

Returns **``cls``** :

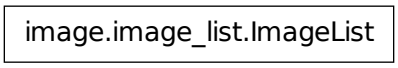
CHAPTER

THIRTYTWO

CORE.IMAGE.IMAGE_LIST

32.1 Module: core.image.image_list

Inheritance diagram for nipy.core.image.image_list:



32.2 ImageList

class ImageList(*images=None*)

__init__(*images=None*)

A lightweight implementation of a list of images.

Parameters images: a sliceable object whose items are meant to be images, :

this is checked by asserting that each has a *coordmap* attribute

>>> from numpy import asarray :

>>> from nipy.testing import funcfile :

>>> from nipy.core.api import Image, ImageList :

>>> from nipy.io.api import load_image :

>>> funcim = load_image(funcfile) :

>>> ilist = ImageList(funcim) :

>>> sublist = ilist[2:5] :

Slicing an ImageList returns a new ImageList :

>>> isinstance(sublist, ImageList) :

True :

Indexing an ImageList returns a new Image :

>>> newimg = ilist[2] :

>>> isinstance(newimg, Image) :

True :

>>> isinstance(newimg, ImageList) :

False :

>>> asarray(sublist).shape :

(3, 20, 2, 20) :

>>> asarray(newimg).shape :

(20, 2, 20) :

next()

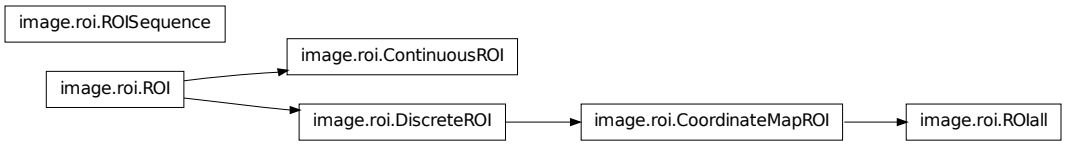
CHAPTER

THIRTYTHREE

CORE.IMAGE.ROI

33.1 Module: core.image.roi

Inheritance diagram for nipy.core.image.roi:



Template region of interest (ROI) module

33.2 Classes

33.2.1 ContinuousROI

class ContinuousROI(*coordinate_system, bfn, args=None, ndim=3*)

Bases: nipy.core.image.roi.ROI

Create an *ROI* with a binary function in a given coordinate system.

__init__(*coordinate_system, bfn, args=None, ndim=3*)

Parameters **coordinate_system** [TODO] TODO

bfn [TODO] TODO

args [TODO] TODO

ndim [int] TODO

tocoordmap(*coordmap*)

Return a *CoordinateMapROI* instance at the voxels in the ROI.

Parameters **coordmap** [TODO] TODO

Returns *CoordinateMapROI*

todiscrete(*voxels*)
Return a *DiscreteROI* instance at the voxels in the ROI.

Parameters **voxels** [TODO] TODO

Returns *DiscreteROI*

33.2.2 CoordinateMapROI

class CoordinateMapROI(*coordinate_system, voxels, coordmap*)
Bases: nipy.core.image.roi.DiscreteROI

__init__(*coordinate_system, voxels, coordmap*)

Parameters **coordinate_system** [TODO] TODO

voxels [TODO] TODO

coordmap [TODO] TODO

mask()

Returns ‘‘numpy.ndarray‘

pool(*image*)
Pool data from an image over the ROI – return fn evaluated at each voxel.

Parameters **image** [*image.Image*] TODO

Returns TODO

Raises valueerror TODO

33.2.3 DiscreteROI

class DiscreteROI(*coordinate_system, voxels*)
Bases: nipy.core.image.roi.ROI

TODO

__init__(*coordinate_system, voxels*)

Parameters **coordinate_system** [TODO] TODO

voxels [TODO] TODO

feature(*fn, **extra*)
Return a feature of an image within the ROI. Feature args are ‘args’, while `extra` are for the readall method. Default is to reduce a ufunc over the ROI. Any other operations should be able to ignore superfluous keywords arguments, i.e. use `extra`.

Parameters **fn** [TODO] TODO

extra [‘‘dict] TODO

Returns *DiscreteROI*

Raises valueerror TODO

Raises notimplementederror TODO

next()

Returns TODO

pool (*fn, **extra*)
Pool data from an image over the ROI – return fn evaluated at each voxel.

Parameters **fn** [TODO] TODO

extras [dict] TODO

Returns TODO

33.2.4 ROI

class ROI (*coordinate_system*)
This is the basic ROI class, which we model as basically a function defined on Euclidean space, i.e. R^3. For practical purposes, this function is evaluated on the range of a Mapping instance.

__init__ (*coordinate_system*)

Parameters **coordinate_system** [TODO] TODO

33.2.5 ROISequence

class ROISequence ()
Bases: list

TODO

__init__ ()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

33.2.6 ROIall

class ROIall (*coordinate_system, voxels, coordmap*)
Bases: nipy.core.image.roi.CoordinateMapROI

An ROI for an entire coordmap. Save time by avoiding compressing, etc.

__init__ (*coordinate_system, voxels, coordmap*)

Parameters **coordinate_system** [TODO] TODO

voxels [TODO] TODO

coordmap [TODO] TODO

mask (*image*)

Parameters **image** [TODO] TODO

Return ‘‘numpy.ndarray‘

pool (*image*)

Parameters image : *image.Image*

Returns None

33.3 Functions

roi_ellipse_fn(*center, form, a=1.0*)

Ellipse determined by regions where a quadratic form is <= a. The quadratic form is given by the inverse of the 'form' argument, so a sphere of radius 10 can be specified as {'form':10**2 * identity(3), 'a':1} or {'form':identity(3), 'a':100}.

Form must be positive definite.

Parameters **form** [TODO] TODO

a [`float`] TODO

Returns TODO

roi_from_array_sampling_coordmap(*data, coordmap*)

Return a *CoordinateMapROI* from an array (data) on a coordmap. interpolation. Obvious ways to extend this.

Parameters **data** [TODO] TODO

coordmap [TODO] TODO

Returns *CoordinateMapROI*

roi_sphere_fn(*center, radius*)

Parameters **center** [TODO] TODO

radius [TODO] TODO

Returns TODO

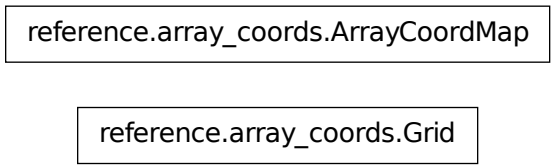
CHAPTER

THIRTYFOUR

CORE.REFERENCE.ARRAY_COORDS

34.1 Module: core.reference.array_coords

Inheritance diagram for nipy.core.reference.array_coords:



Some CoordinateMaps have input_coords that are 'array' coordinates, hence the function of the CoordinateMap can be evaluated at these 'array' points.

This module tries to make these operations easier by defining a class ArrayCoordMap that is essentially a CoordinateMap and a shape.

This class has two properties: values, transposed_values the CoordinateMap at np.indices(shape).

The class Grid is meant to take a CoordinateMap and an np.mgrid-like notation to create an ArrayCoordMap.

34.2 Classes

34.2.1 ArrayCoordMap

class ArrayCoordMap(*coordmap, shape*)

Bases: object

When the input_coords of a CoordinateMap can be thought of as 'array' coordinates, i.e. an 'input_shape' makes sense. We can than evaluate the CoordinateMap at np.indices(input_shape)

__init__(*coordmap, shape*)

Parameters **coordmap** : CoordinateMap

A CoordinateMap with input_coords that are 'array' coordinates.

shape : sequence of int

The size of the (implied) underlying array.

static **from_shape**(*coordmap, shape*)
Create an evaluator assuming that coordmap.input_coords are 'array' coordinates.

transposed_values
Get values of ArrayCoordMap in an array of shape (self.coordmap.ndim[1],) + self.shape)

values
Get values of ArrayCoordMap in a 2-dimensional array of shape (product(self.shape), self.coordmap.ndim[1]))

34.2.2 Grid

class Grid(*coords*)
Bases: object

Simple class to construct Affine instances with slice notation like np.ogrid/np.mgrid.

```
>>> c = CoordinateSystem('xy', 'input')
>>> g = Grid(c)
>>> points = g[-1:1:21j,-2:4:31j]
>>> points.coordmap.affine
array([[ 0.1,  0. , -1. ],
       [ 0. ,  0.2, -2. ],
       [ 0. ,  0. ,  1. ]])


>>> print points.coordmap.input_coords
name: 'product', coord_names: ('i0', 'i1'), coord_dtype: float64
>>> print points.coordmap.output_coords
name: 'input', coord_names: ('x', 'y'), coord_dtype: float64


>>> points.shape
(21, 31)
>>> print points.transposed_values.shape
(2, 21, 31)
>>> print points.values.shape
(651, 2)
```

__init__(*coords*)
Initialize Grid object

Parameters coords: ''CoordinateMap'' or ''CoordinateSystem'' :

A coordinate map to be 'sliced' into. If coords is a CoordinateSystem, then an Affine instance is created with coords with identity transformation.

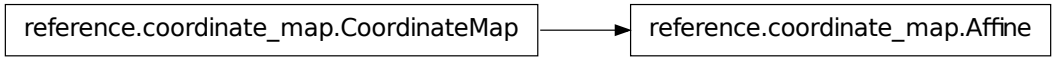
CHAPTER

THIRTYFIVE

CORE.REFERENCE.COORDINATE_MAP

35.1 Module: core.reference.coordinate_map

Inheritance diagram for nipy.core.reference.coordinate_map:



CoordinateMaps map (transform) an image from an input space to an output space.

A CoordinateMap object contains all the details about an input CoordinateSystem, an output CoordinateSystem and the mappings between them. The *mapping* transforms an image from the input coordinate system to the output coordinate system. And the *inverse_mapping* performs the opposite transformation. The *inverse_mapping* can be specified explicity when creating the CoordinateMap or implicitly in the case of an Affine CoordinateMap.

35.2 Classes

35.2.1 Affine

class Affine (*affine, input_coords, output_coords*)

Bases: nipy.core.reference.coordinate_map.CoordinateMap

A class representing an affine transformation from an input coordinate system to an output coordinate system.

This class has an affine property, which is a matrix representing the affine transformation in homogeneous coordinates. This matrix is used to perform mappings, rather than having an explicit mapping function.

```
>>> inp_cs = CoordinateSystem('ijk')
>>> out_cs = CoordinateSystem('xyz')
>>> cm = Affine(np.diag([1, 2, 3, 1]), inp_cs, out_cs)
>>> cm.affine
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  2.,  0.,  0.],
       [ 0.,  0.,  3.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

```
>>> cm([1,1,1])
array([[ 1.,  2.,  3.]])
>>> icm = cm.inverse
>>> icm([1,2,3])
array([[ 1.,  1.,  1.]])
```

__init__ (*affine, input_coords, output_coords*)

Return an CoordinateMap specified by an affine transformation in homogeneous coordinates.

Parameters **affine** : array-like

affine homogenous coordinate matrix

input_coords : `CoordinateSystem`

input coordinates

output_coords : `CoordinateSystem`

output coordinates

Notes

The dtype of the resulting matrix is determined by finding a safe typecast for the input_coords, output_coords and affine.

affine

The affine transform matrix of the Affine CoordinateMap.

copy()

Create a copy of the coordmap.

Returns **cm** : *CoordinateMap*

Examples

```
>>> cm = Affine(np.eye(4), CoordinateSystem('ijk'), CoordinateSystem('xyz'))
>>> cm_copy = cm.copy()
>>> cm is cm_copy
False
```

Note that the matrix (affine) is not a pointer to the same data, it's a full independent copy

```
>>> cm.affine[0,0] = 2.0
>>> cm_copy.affine[0,0]
1.0
```

static **from_params** (*innames, outnames, params*)

Create an *Affine* instance from sequences of innames and outnames.

Parameters **innames** : `tuple` of `string`

The names of the axes of the input coordinate systems

outnames : `tuple` of `string`

The names of the axes of the output coordinate systems

params : *Affine*, *ndarray* or *(ndarray, ndarray)*

An affine mapping between the input and output coordinate systems. This can be represented either by a single ndarray (which is interpreted as the representation of the mapping in homogeneous coordinates) or an (A,b) tuple.

Returns **aff** : *Affine* object instance

Notes

Precondition `len(shape) == len(names)`

Raises valueerror `if len(shape) != len(names)`

static **from_start_step**(*innames, outnames, start, step*)
Create an *Affine* instance from sequences of names, start and step.

Parameters **innames** : `tuple` of `string`

The names of the axes of the input coordinate systems

outnames : `tuple` of `string`

The names of the axes of the output coordinate systems

start : `tuple` of `float`

Start vector used in constructing affine transformation

step : `tuple` of `float`

Step vector used in constructing affine transformation

Returns **cm** : *CoordinateMap*

Notes

`len(names) == len(start) == len(step)`

Examples

```
>>> cm = Affine.from_start_step('ijk', 'xyz', [1, 2, 3], [4, 5, 6])
>>> cm.affine
array([[ 4.,  0.,  0.,  1.],
       [ 0.,  5.,  0.,  2.],
       [ 0.,  0.,  6.,  3.],
       [ 0.,  0.,  0.,  1.]])
```

static **identity**(*names*)
Return an identity coordmap of the given shape.

Parameters **names** : `tuple` of `string`

Names of Axes in output CoordinateSystem

Returns **cm** : *CoordinateMap*

`CoordinateMap` with *CoordinateSystem* input and an identity transform, with identical input and output coords.

Examples

```
>>> cm = Affine.identity('ijk')
>>> cm.affine
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> print cm.input_coords
name: 'input', coord_names: ('i', 'j', 'k'), coord_dtype: float64
>>> print cm.output_coords
name: 'output', coord_names: ('i', 'j', 'k'), coord_dtype: float64
```

inverse
Return the inverse coordinate map.

inverse_mapping
The inverse affine mapping from the Affine CoordinateMap.

35.2.2 CoordinateMap

class CoordinateMap(*mapping, input_coords, output_coords, inverse_mapping=None*)
Bases: object

A set of input and output CoordinateSystems and a mapping between them.

For example, the mapping may represent the mapping of an image from voxel space (the input coordinates) to real space (the output coordinates). The mapping may be an affine or non-affine transformation.

Attributes **input_coords** : CoordinateSystem

The input coordinate system.

output_coords : CoordinateSystem

The output coordinate system.

mapping : callable

A callable that maps the input_coords to the output_coords.

inverse_mapping : None or callable

A callable that maps the output_coords to the input_coords. Not all mappings have an inverse, in which case inverse_mapping is None.

Examples

```
>>> input_coords = CoordinateSystem('ijk', 'voxels')
>>> output_coords = CoordinateSystem('xyz', 'world')
>>> mni_orig = np.array([-90.0, -126.0, -72.0])
>>> mapping = lambda x: x + mni_orig
>>> inv_mapping = lambda x: x - mni_orig
>>> cm = CoordinateMap(mapping, input_coords, output_coords, inv_mapping)
```

Map the first 3 voxel coordinates, along the x-axis, to mni space:

```
>>> x = np.array([[0,0,0], [1,0,0], [2,0,0]])
>>> cm.mapping(x)
array([[ -90., -126.,  -72.],
       [ -89., -126.,  -72.],
       [ -88., -126.,  -72.]])
```

__init__(*mapping, input_coords, output_coords, inverse_mapping=None*)

Create a CoordinateMap given the input/output coords and mappings.

Parameters **mapping** : callable

The mapping between input and output coordinates

input_coords : `CoordinateSystem`

The input coordinate system

output_coords : `CoordinateSystem`

The output coordinate system

inverse_mapping : None or callable, optional

The optional inverse of mapping, with the intention being `x = inverse_mapping(mapping(x))`. If the mapping is affine and invertible, then this is true for all x. The default is None

Returns **coordmap** : CoordinateMap

copy()

Create a copy of the coordmap.

Returns **coordmap** : CoordinateMap

input_coords

input coordinate system

inverse

Return a new CoordinateMap with the mappings reversed

inverse_mapping

The mapping from output_coords to input_coords

mapping

The mapping from input_coords to output_coords.

ndim

Number of dimensions of input and output coordinates.

output_coords

output coordinate system

35.3 Functions

compose(**cmaps*)

Return the composition of two or more CoordinateMaps.

linearize(*mapping, ndimin, step=1, origin=None, dtype=None*)

Given a Mapping of ndimin variables, return the linearization of mapping at origin based on a given step size in each coordinate axis.

If not specified, origin defaults to np.zeros(ndimin, dtype=dtype).

Parameters **mapping** : callable

A function to linearize

ndimin : int

Number of input dimensions to mapping

step : scalar, optional

step size over which to calculate linear components. Default 1

origin : None or array, optional

Origin at which to linearize mapping. If None, origin is `np.zeros(ndimin)`

dtype : None or np.dtype, optional

dtype for return. Default is None. If `dtype` is None, and `step` is an ndarray, use `step.dtype`. Otherwise use np.float.

Returns **C** : array

Linearization of mapping in homogeneous coordinates, i.e. an array of size (ndimout+1, ndimin+1) where ndimout = mapping(origin).shape[0].

product(**cmaps*)
Return the "topological" product of two or more CoordinateMaps.

reorder_input(*coordmap, order=None*)
Create a new coordmap with reversed input_coords. Default behaviour is to reverse the order of the input_coords. If the coordmap has a shape, the resulting one will as well.

reorder_output(*coordmap, order=None*)
Create a new coordmap with reversed output_coords. Default behaviour is to reverse the order of the input_coords.

replicate(*coordmap, n, concataxis='concat'*)
Create a CoordinateMap by taking the product of coordmap with a 1-dimensional 'concat' CoordinateSystem

Parameters **coordmap** [*CoordinateMap*] The coordmap to be used

n [`int`] The number of tiems to concatenate the coordmap

concataxis [`string`] The name of the new dimension formed by concatenation

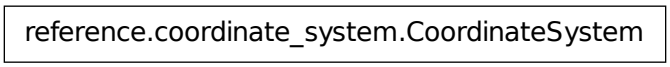
CHAPTER

THIRTYSIX

CORE.REFERENCE.COORDINATE_SYSTEM

36.1 Module: core.reference.coordinate_system

Inheritance diagram for nipy.core.reference.coordinate_system:



CoordinateSystems are used to represent the space in which the image resides.

A CoordinateSystem contains named coordinates, one for each dimension and a coordinate dtype. The purpose of the CoordinateSystem is to specify the name and order of the coordinate axes for a particular space. This allows one to compare two CoordinateSystems to determine if they are equal.

36.2 Class

36.3 CoordinateSystem

class CoordinateSystem(*coord_names, name='', coord_dtype=<type 'float'>*)

Bases: object

An ordered sequence of named coordinates of a specified dtype.

A coordinate system is defined by the names of the coordinates, (attribute coord_names) and the numpy dtype of each coordinate value (attribute coord_dtype). The coordinate system can also have a name.

```
>>> names = ['first', 'second', 'third']
>>> cs = CoordinateSystem(names, 'a coordinate system', np.float)
>>> cs.coord_names
('first', 'second', 'third')
>>> cs.name
'a coordinate system'
>>> cs.coord_dtype
dtype('float64')
```

The coordinate system also has a `dtype` which is the composite numpy dtype, made from the (`names`, `coord_dtype`).

```
>>> dtype_template = [(name, np.float) for name in cs.coord_names]
>>> dtype_should_be = np.dtype(dtype_template)
>>> cs.dtype == dtype_should_be
True
```

Two CoordinateSystems are equal if they have the same dtype. The CoordinateSystem names may be different.

```
>>> another_cs = CoordinateSystem(names, 'irrelevant', np.float)
>>> cs == another_cs
True
>>> cs.dtype == another_cs.dtype
True
>>> cs.name == another_cs.name
False
```

__init__(*coord_names, name='', coord_dtype=<type 'float'>*)

Create a coordinate system with a given name and coordinate names.

The CoordinateSystem has two dtype attributes:

1. self.coord_dtype is the dtype of the individual coordinate values
2. self.dtype is the recarray dtype for the CoordinateSystem which combines the coord_names and the coord_dtype. This functions as the description of the CoordinateSystem.

Parameters **coord_names** : iterable

A sequence of coordinate names.

name : string, optional

The name of the coordinate system

coord_dtype : np.dtype, optional

The dtype of the coord_names. This should be a built-in numpy scalar dtype. (default is np.float). The value can by anything that can be passed to the np.dtype constructor. For example `np.float`, `np.dtype(np.float)` or `f8` all result in the same `coord_dtype`.

Examples

```
>>> c = CoordinateSystem('ij', name='input')
>>> print c
name: 'input', coord_names: ('i', 'j'), coord_dtype: float64
```

```
>>> c.coord_dtype
dtype('float64')
```

coord_dtype

The dtype of the coordinates in the CoordinateSytem

coord_names

The coordinate names in the CoordinateSystem

dtype
The dtype of the CoordinateSystem.

index(*coord_name*)
Return the index of a given named coordinate.

```
>>> c = CoordinateSystem('ij', name='input')
>>> c.index('i')
0
```

```
>>> c.index('j')
1
```

ndim
The number of coordinates in the CoordinateSystem

36.4 Functions

product(**coord_systems*)
Create the product of a sequence of CoordinateSystems.

The coord_dtype of the result will be determined by `safe_dtype`.

Parameters **coord_systems** : sequence of `CoordinateSystem`

Returns **product_coord_system** : `CoordinateSystem`

Examples

```
>>> c1 = CoordinateSystem('ij', 'input', coord_dtype=np.float32)
>>> c2 = CoordinateSystem('kl', 'input', coord_dtype=np.complex)
>>> c3 = CoordinateSystem('ik', 'in3')
```

```
>>> print product(c1,c2)
name: 'product', coord_names: ('i', 'j', 'k', 'l'), coord_dtype: complex128
```

```
>>> product(c2,c3)
Traceback (most recent call last):
   ...
ValueError: coord_names must have distinct names
>>>
```

safe_dtype(**dtypes*)
Determine a dtype to safely cast all of the given dtypes to.

Safe dtypes are valid numpy dtypes or python types which can be cast to numpy dtypes. See numpy.sctypes for a list of valid dtypes. Composite dtypes and string dtypes are not safe dtypes.

To see if your dtypes are valid, build a numpy array of each dtype and the resulting object should return *1* from the varname.dtype.isbuiltin attribute.

Parameters **dtypes** : sequence of builtin `np.dtype`

Returns **dtype** : np.dtype

Examples

```
>>> c1 = CoordinateSystem('ij', 'input', coord_dtype=np.float32)
>>> c2 = CoordinateSystem('kl', 'input', coord_dtype=np.complex)
>>> safe_dtype(c1.coord_dtype, c2.coord_dtype)
dtype('complex128')


>>> # Strings are invalid dtypes
>>> safe_dtype(type('foo'))
...
TypeError: dtype must be valid numpy dtype int, uint, float or complex


>>> # Check for a valid dtype
>>> myarr = np.zeros(2, np.float32)
>>> myarr.dtype.isbuiltin
1


>>> # Composite dtypes are invalid
>>> mydtype = np.dtype([('name', 'S32'), ('age', 'i4')])
>>> myarr = np.zeros(2, mydtype)
>>> myarr.dtype.isbuiltin
0
>>> safe_dtype(mydtype)
...
TypeError: dtype must be valid numpy dtype int, uint, float or complex
```

CHAPTER

THIRTYSEVEN

CORE.REFERENCE.SLICES

37.1 Module: `core.reference.slices`

A set of methods to get coordinate maps which represent slices in space.

37.2 Functions

`bounding_box`(*coordmap, shape*)
Determine a valid bounding box from a CoordinateMap instance.

Parameters coordmap : *CoordinateMap*

`from_origin_and_columns`(*origin, colvectors, shape, output_coords*)
Return a CoordinateMap representing a slice based on a given origin, a pair of direction vectors which span the slice, and a shape.

Parameters **origin** [the corner of the output coordinates, i.e. the [0]*ndimin] point

colvectors : the steps in each voxel direction shape : how many steps in each voxel direction output_coords : a CoordinateSystem for the output

Returns *CoordinateMap*

`xslice`(*x, zlim, ylim, output_coords, shape*)
Return a slice through a 3d box with x fixed.

Parameters **y** [TODO] TODO

zlim [TODO] TODO

ylim [TODO] TODO

xlim [TODO] TODO

shape [TODO] TODO

output_coords [TODO] TODO

`yslice`(*y, zlim, xlim, output_coords, shape*)
Return a slice through a 3d box with y fixed.

Parameters **x** [TODO] TODO

zlim [TODO] TODO

ylim [TODO] TODO

xlim [TODO] TODO

shape [TODO] TODO

output_coords [TODO] TODO

zslice(*z, ylim, xlim, output_coords, shape*)
Return a slice through a 3d box with z fixed.

Parameters **z** [TODO] TODO

ylim [TODO] TODO

xlim [TODO] TODO

shape [TODO] TODO

output_coords [TODO] TODO

CHAPTER

THIRTYEIGHT

CORE.TRANSFORMS.AFFINES

38.1 Module: core.transforms.affines

Functions working on affine transformation matrices.

38.2 Functions

from_matrix_vector(*matrix, vector*)

Combine a matrix and vector into a homogeneous transform.

Combine a rotation matrix and translation vector into a transform in homogeneous coordinates.

Parameters **matrix** : ndarray

An NxN array representing the rotation matrix.

vector : ndarray

A 1xN array representing the translation.

Returns **xform** : ndarray

An N+1xN+1 transform matrix.

See Also:

to_matrix_vector

to_matrix_vector(*transform*)

Split a transform into it's matrix and vector components.

The tranformation must be represented in homogeneous coordinates and is split into it's rotation matrix and translation vector components.

Parameters **transform** : ndarray

Transform matrix in homogeneous coordinates. Example, a 4x4 transform representing rotations and translations in 3 dimensions.

Returns **matrix, vector** : ndarray

The matrix and vector components of the transform matrix. For an NxN transform, matrix will be N-1xN-1 and vector will be 1xN-1.

See Also:

from_matrix_vector

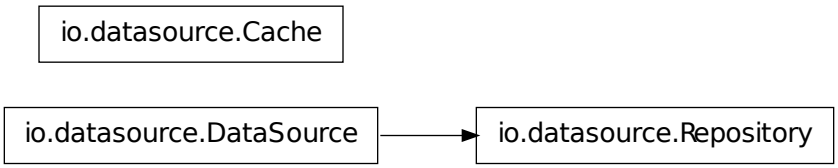
CHAPTER

THIRTYNINE

IO.DATASOURCE

39.1 Module: io.datasource

Inheritance diagram for nipy.io.datasource:



39.2 Classes

39.2.1 Cache

class Cache(*cachepath=None*)

Bases: object

A file cache. The path of the cache can be specified or else use ~/.nipy/cache by default.

__init__(*cachepath=None*)

cache(*uri*)

Copy a file into the cache.

Returns None

clear()

Delete all files in the cache.

Returns None

filename(*uri*)

Return the complete path + filename within the cache.

Returns string

filepath(*uri*)
Return the complete path + filename within the cache.

iscached(*uri*)
Check if a file exists in the cache.

Returns `bool`

retrieve(*uri*)
Retrieve a file from the cache. If not already there, create the file and add it to the cache.

Returns `file`

tempfile(*suffix=", prefix="*)
Return an temporary file name in the cache

39.2.2 DataSource

class DataSource(*cachepath='.'*)
Bases: `object`

__init__(*cachepath='.'*)

cache(*pathstr*)

exists(*pathstr*)

filename(*pathstr*)

open(*pathstr, mode='r'*)

tempfile(*suffix=", prefix="*)
Return an temporary file name in the cache

39.2.3 Repository

class Repository(*baseurl, cachepath=None*)
Bases: `nipy.io.datasource.DataSource`

DataSource with an implied root.

__init__(*baseurl, cachepath=None*)

exists(*pathstr*)

filename(*pathstr*)

open(*pathstr, mode='r'*)

39.3 Functions

ensuredirs(*directory*)
Ensure that the given directory path actually exists. If it doesn't, create it.

Returns `None`

isurl(*pathstr*)
Check whether a given string can be parsed as a URL.

Parameters ***pathstr*** [string] The string to be checked.

Returns `bool`

iswritemode(*mode*)
Test if the given mode will open a file for writing.

Parameters ***mode*** [string] The mode to be checked

Returns `bool`

iszip(*filename*)
Is this filename a zip file.

Returns `bool`

splitzipext(*filename*)
return (base, zip_extension) from filename. If filename does not have a zip extention then base = filename and zip_extension = None

unzip(*filename*)
Unzip the given file into another file. Return the new file's name.

Returns `string`

CHAPTER

FORTY

IO.FILES

40.1 Module: `io.files`

The image module provides basic functions for working with images in nipy. Functions are provided to load, save and create image objects, along with iterators to easily slice through volumes.

load : load an image from a file

save : save an image to a file

fromarray : create an image from a numpy array

40.1.1 Examples

See documentation for load and save functions for 'working' examples.

40.2 Functions

`coerce2nifti`(*img, standard=False*)

Coerce an Image into a new Image with a valid NIFTI coordmap so that it can be saved.

If standard is True, the resulting image has 'standard'-ordered input_coordinates, i.e. 'ijklmno'[:img.ndim]

`load`(*filename, mode='r'*)

Load an image from the given filename.

Load an image from the file specified by `filename`.

Parameters **filename** : string

Should resolve to a complete filename path.

mode : Either 'r' or 'r+'

Returns **image** : An *Image* object

If successful, a new *Image* object is returned.

See Also:

`save_image` function for saving images

`fromarray` function for creating images from numpy arrays

Examples

```
>>> from nipy.io.api import load_image
>>> from nipy.testing import anatfile
>>> img = load_image(anatfile)
>>> img.shape
(25, 35, 25)
```

save(*img, filename, dtype=None*)

Write the image to a file.

Parameters **img** : An *Image* object

filename : string

Should be a valid filename.

Returns **image** : An *Image* object

See Also:

load_image function for loading images

fromarray function for creating images from numpy arrays

Notes

Filetype is determined by the file extension in 'filename'. Currently the following filetypes are supported:

Nifti single file : ['.nii', '.nii.gz'] Nifti file pair : ['.hdr', '.hdr.gz'] Analyze file pair : ['.img', 'img.gz']

Examples

```
>>> import os
>>> import numpy as np
>>> from tempfile import mkstemp
>>> from nipy.core.api import fromarray
>>> from nipy.io.api import save_image
>>> data = np.zeros((91,109,91), dtype=np.uint8)
>>> img = fromarray(data, 'kji', 'zxy')
>>> fd, name = mkstemp(suffix='.nii.gz')
>>> tmpfile = open(name)
>>> saved_img = save_image(img, tmpfile.name)
>>> saved_img.shape
(91, 109, 91)
>>> tmpfile.close()
>>> os.unlink(name)
```

CHAPTER

FORTYONE

IO.NIFTI_REF

41.1 Module: `io.nifti_ref`

An implementation of the dimension info as desribed in

http://nifti.nimh.nih.gov/pub/dist/src/niftilib/nifti1.h

In particular, it allows one to check if a *CoordinateMap* instance can be coerced into a valid NIFTI CoordinateMap instance. For a valid NIFTI coordmap, we can then ask which axes correspond to time, slice, phase and frequency.

41.1.1 Axes:

NIFTI files can have up to seven dimensions. We take the convention that the output coordinate names are ['x','y','z','t','u','v','w'] and the input coordinate names are ['i','j','k','l','m','n','o'].

In the NIFTI specification, the order of the output coordinates (at least the first 3) are fixed to be ['x','y','z'] and their order is not meant to change. As for input coordinates, the first three can be reordered, so ['j','k','i','l'] is valid, for instance.

NIFTI has a 'diminfo' header attribute that optionally specifies the order of the ['i', 'j', 'k'] axes. To use similar terms to those in the nifti1.h header, 'phase' corresponds to 'i'; 'frequency' to 'j' and 'slice' to 'k'. We use ['i','j','k'] instead because there are images for which the terms 'phase' and 'frequency' have no proper meaning. See the functions *get_freq_axes*, *get_phase_axis* for how this is dealt with.

41.1.2 Voxel coordinates:

NIFTI's voxel convention is what can best be described as 0-based FORTRAN indexing (confirm this). For example: suppose we want the x=20-th, y=10-th pixel of the third slice of an image with 30 64x64 slices. This

```
>>> from nipy.testing import anatfile
>>> from nipy.io.api import load_image
>>> nifti_ijk = [19,9,2]
>>> fortran_ijk = [20,10,3]
>>> c_kji = [2,9,19]
>>> imgarr = np.asarray(load_image(anatfile))
>>> request1 = imgarr[nifti_ijk[2], nifti_ijk[1], nifti_ijk[0]]
>>> request2 = imgarr[fortran_ijk[2]-1,fortran_ijk[1]-1, fortran_ijk[0]-1]
>>> request3 = imgarr[c_kji[0],c_kji[1],c_kji[2]]
>>> request1 == request2
True
```

```
>>> request2 == request3
True
```

FIXME: (finish this thought.... Are we going to open NIFTI files with NIFTI input coordinates?) For this reason, we have to consider whether we should transpose the memmap from pynifti.

41.2 Functions

coerce_coordmap(*coordmap*)
Determine if a given CoordinateMap instance can be used as a valid coordmap for a NIFTI image, so that an Image can be saved.

If the input coordinates must be reordered, the order defaults to the NIFTI order ['i','j','k','l','m','n','o'].

coordmap4io(*coordmap*)
Create a valid coordmap for saving with a NIFTI file. Also returns the NIFTI diminfo and pixdim header attributes.

coordmap_from_ioimg(*affine, diminfo, pixdim, shape*)
Generate a CoordinateMap from an affine transform.

This is a convenience function to create a CoordinateMap from image attributes. It uses the orientation field from pynifti IO to map to the nipy *names*, prepending *time* or *vector* depending on dimension.

FIXME: This is an internal function and should be revisited when the CoordinateMap is refactored.

get_diminfo(*coordmap*)
Get diminfo byte from a coordmap, after validating it as a valid NIFTI coordmap.

get_freq_axis(*coordmap*)
Determine the freq axis of a valid NIFTI coordmap.

get_phase_axis(*coordmap*)
Determine the freq axis of a valid NIFTI coordmap.

get_pixdim(*coordmap, full_length=False*)
Get pixdim from a coordmap, after validating it as a valid NIFTI coordmap. The pixdims are taken from the output_coords. Specifically, for each axis 'xyztuvw', if the corresponding output_coord has a step, use it, otherwise use 0.

get_slice_axis(*coordmap*)
Determine the slice axis of a valid NIFTI coordmap.

get_time_axis(*coordmap*)
Determine the time axis of a valid NIFTI coordmap.

ijk_from_diminfo(*diminfo*)
Determine the order of the 'ijk' dimensions from the diminfo byte of a NIFTI header. If any of them are 'undefined', set the order alphabetically, after having set the ones that are defined.

iscoerceable(*coordmap*)
Determine if a given CoordinateMap instance can be used as a valid coordmap for a NIFTI image, so that an Image can be saved.

This may raise various warnings about the coordmap.

standard_order(*coordmap*)
Take a valid NIFTI coordmap, and return a coordmap with input_coordinates in the standard order, i.e. with names 'ijklmno'[:coordmap.ndim[0]] and the order of coordinates that put the coordmap into standard order.

NOTE: If the coordmap is only 'coerceable' and not 'valid' (i.e. warnings are raised, then this may give unexpected results because it only checks the reordering of the first 3 coordinates.

```
>>> cmap = Affine.from_params('ikjl', 'xyzt', np.identity(5))
>>> sorder, scmap = standard_order(cmap)
>>> print cmap.input_coords.coord_names
('i', 'k', 'j', 'l')
>>> print scmap.input_coords.coord_names
('i', 'j', 'k', 'l')
>>> print scmap.output_coords.coord_names
('x', 'y', 'z', 't')
>>> print cmap.output_coords.coord_names
('x', 'y', 'z', 't')
```

CHAPTER

FORTYTWO

IO.PYNIFTIIO

42.1 Module: io.pyniftiio

Inheritance diagram for nipy.io.pyniftiio:

The Nipy interface to the PyNifti library.

Use PyNifti to open files and extract necessary data to generate a Nipy Image.

42.2 PyNiftiIO

class PyNiftiIO(*data, mode='r', dtype=None, header={}*)

Bases: object

Wrapper around the PyNifit image class.

__init__(*data, mode='r', dtype=None, header={}*)

Create a PyNiftiIO object.

Parameters **data** : {array_like, filename}

Data should be either a filename (string), a numpy array or an object that implements the __array__ interface from which we get an array.

mode : {'r', 'w'}, optional

File access mode. Read-only or read-write mode.

dtype : numpy.dtype

The dtype to save the data array as. An exception is raised if the requested dtype is not a valid nifti data type.

Returns **pyniftiio** : A PyNiftiIO object

affine

diminfo

filename

header
Get or set the pynifti header.

ndim

orientation

pixdim

save(*affine, pixdim, diminfo, filename=None*)

shape

getaffine(*img*)
Get affine transform from a NiftiImage.

Parameters **img** : NiftiImage

image opened with PyNifti

Returns **affine** : array

The 4x4 affine transform as a numpy array

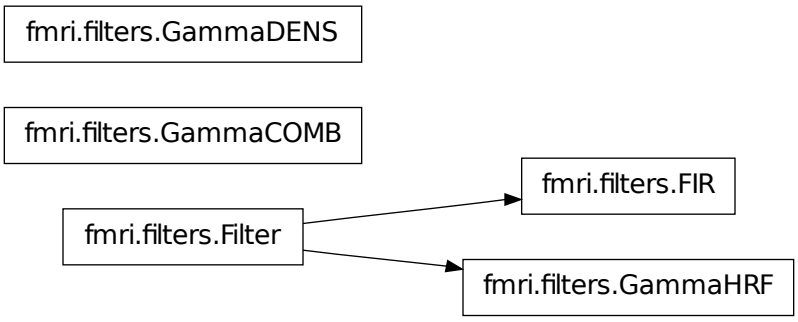
CHAPTER

FORTYTHREE

MODALITIES.FMRI.FILTERS

43.1 Module: modalities.fmri.filters

Inheritance diagram for nipy.modalities.fmri.filters:



TODO

43.2 Classes

43.2.1 FIR

class FIR (*parameters*)

Bases: nipy.modalities.fmri.filters.Filter

A class for FIR filters: i.e. the filter is a collection of square waves. Parameters (start and duration) are specified as a kx2 matrix for k square waves.

```
>>> GUI = True
>>> from nipy.modalities.fmri import filters
>>> from pylab import *
>>> from numpy import *
>>> parameters = array([[1., 2.], [2., 5.], [4., 8.]])
>>> IRF = filters.FIR(parameters)
```

```
>>> _ = plot(arange(0, 15, 0.1), sum(IRF(arange(0, 15, 0.1)), axis=0))
>>> ylab = ylabel('Filters')
>>> xlab = xlabel('Time (s)')
>>> show()
```

__init__ (*parameters*)

Parameters ***parameters*** [TODO] TODO

43.2.2 Filter

class Filter (*IRF, names, dt=0.02, tmin=-10.0, tmax=500.0*)

Bases: object

Takes a list of impulse response functions (IRFs): main purpose is to convolve a functions with each IRF for Design. The class assumes the range of the filter is effectively 50 seconds, can be changed by setting tmax – this is just for the __mul__ method for convolution.

__init__ (*IRF, names, dt=0.02, tmin=-10.0, tmax=500.0*)

Parameters ***IRF*** [TODO] TODO

names [TODO] TODO

dt [float] TODO

tmin [float] TODO

tmax [float] TODO

convolve (*fn, interval=None, dt=None*)

Take a (possibly vector-valued) function fn of time and return a linearly interpolated function after convolving with the filter.

43.2.3 GammaCOMB

class GammaCOMB (*fns*)

TODO

__init__ (*fns*)

Parameters ***fns*** [TODO] TODO

deriv (*const=1.0*)

Parameters ***const*** [float] TODO

Returns *GammaCOMB*

43.2.4 GammaDENS

class GammaDENS (*alpha, nu, coef=1.0*)

A class for a Gamma density which knows how to differentiate itself.

By default, normalized to integrate to 1.

__init__ (*alpha, nu, coef=1.0*)

Parameters ***alpha*** [TODO] TODO

nu [TODO] TODO

coef [float] TODO

deriv(*const=1.0*)

Differentiate a Gamma density. Returns a GammaCOMB that can evaluate the derivative.

43.2.5 GammaHRF

class GammaHRF(*parameters*)

Bases: nipy.modalities.fmri.filters.Filter

A class that represents the Gamma basis in SPM: i.e. the filter is a collection of a certain number of Gamma densities. Parameters are specified as a kx2 matrix for k Gamma functions.

__init__(*parameters*)

Parameters ***parameters*** [TODO] TODO

deriv(*const=1.0*)

Parameters ***const*** [float] TODO

Returns TODO

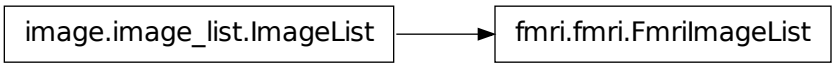
CHAPTER

FORTYFOUR

MODALITIES.FMRI.FMRI

44.1 Module: modalities.fmri.fmri

Inheritance diagram for nipy.modalities.fmri.fmri:



44.2 Class

44.3 FmriImageList

class FmriImageList(*images=None, volume_start_times=None, slice_times=None*)

Bases: nipy.core.image.image_list.ImageList

Class to implement image list interface for FMRI time series

Allows metadata such as volume and slice times

__init__()

A lightweight implementation of an fMRI image as in ImageList

Parameters images: a sliceable object whose items are meant to be images, :

this is checked by asserting that each has a *coordmap* attribute

volume_start_times: start time of each frame. It can be specified :

either as an ndarray with len(images) elements or as a single float, the TR. Defaults to arange(len(images)).astype(np.float)

slice_times: ndarray specifying offset for each slice of each frame :

44.4 Functions

fmri_generator(*data, iterable=None*)
This function takes an iterable object and returns a generator for

[numpy.asarray(data)[:,item] for item in iterator]

This is used to get time series out of a 4d fMRI image.

Note that if data is an FmriImageList instance, there is more overhead involved in calling numpy.asarray(data) than if data is in Image instance.

If iterables is None, it defaults to range(data.shape[0])

fromimage(*fourdimage, volume_start_times=None, slice_times=None*)
Create an FmriImageList from a 4D Image.

Load an image from the file specified by `url` and `datasource`.

Note this assumes that the 4d Affine mapping is such that it can be made into a list of 3d Affine mappings

Parameters **fourdimage: a 4D Image** :

volume_start_times: start time of each frame. It can be specified :

either as an ndarray with len(images) elements or as a single float, the TR. Defaults to the diagonal entry of slowest moving dimension of Affine transform

slice_times: ndarray specifying offset for each slice of each frame :

TODO: watch out for reordering the output coordinates to (x,y,z,t) :

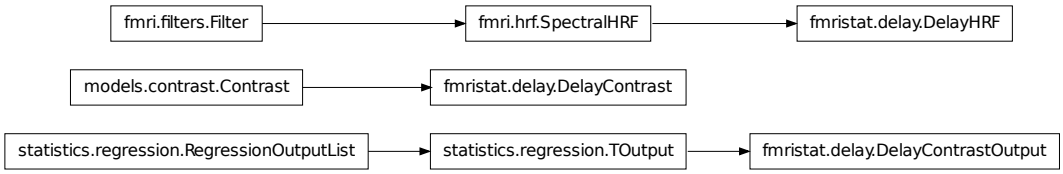
CHAPTER

FORTYFIVE

MODALITIES.FMRI.FMRISTAT.DELAY

45.1 Module: modalities.fmri.fmristat.delay

Inheritance diagram for nipy.modalities.fmri.fmristat.delay:



This module defines a class to output estimates of delays and contrasts of delays.

Liao, C.H., Worsley, K.J., Poline, J-B., Aston, J.A.D., Duncan, G.H., Evans, A.C. (2002). 'Estimating the delay of the response in fMRI data.' NeuroImage, 16:593-606.

45.2 Classes

45.2.1 DelayContrast

class DelayContrast (*fns, weights, formula, IRF=None, name='', rownames=, []*)

Bases: nipy.fixes.scipy.stats.models.contrast.Contrast

Specify a delay contrast.

Delay contrasts are specified by a sequence of functions and weights, the functions should NOT already be convolved with any HRF. They will be convolved with self.IRF which is expected to be a filter with a canonical HRF and its derivative – defaults to the Glover model.

Weights should have the same number of columns as len(fns), with each row specifying a different contrast.

__init__ (*fns, weights, formula, IRF=None, name='', rownames=, []*)

Parameters ***fns*** [TODO] TODO

weights [TODO] TODO

formula [TODO] TODO

IRF [TODO] TODO

name [string] TODO

rownames [[string]] TODO

compute_matrix(*time=None*)

Parameters ***time*** [TODO] TODO

Returns `None`

extract(*results*)

Parameters ***results*** [TODO] TODO

Returns *ContrastResults*

isestimable(*t*)

To estimate the delay, it is assumed that the response contains

(f ** HRF)(t + delta)

for each delay model time series 'f'. More specifically, it is assumed that

f(t + delta) = c1 * (f ** HRF)(t) + delta * c2 * (f ** dHRF)(t)

where HRF and dHRF are the HRFs for this delay contrast.

This function checks to ensure that the columns

[(f ** HRF)(t), (f ** dHRF(t))]

are in the column space of the fMRI regression model.

Parameters ***t*** [TODO] TODO

Returns `None`

Raises valueerror if any of the columns are not in the column space of the model

45.2.2 DelayContrastOutput

class DelayContrastOutput(*coordmap, contrast, IRF=None, dt=0.01, delta=None, subpath='delays', clobber=False, path='.', ext='.hdr', volume_start_times=, [], **kw*)

Bases: `nipy.algorithms.statistics.regression.TOutput`

TODO

__init__(*coordmap, contrast, IRF=None, dt=0.01, delta=None, subpath='delays', clobber=False, path='.', ext='.hdr', volume_start_times=, [], **kw*)

Parameters ***coordmap*** [TODO] TODO

contrast [TODO] TODO

IRF [TODO] TODO

dt [float] TODO

delta [TODO] TODO

subpath [string] TODO

clobber [bool] TODO

path [string] TODO

ext [string] TODO

volume_start_times [TODO] TODO

kw [dict] Passed through to the constructor of *TContrastOutput*

extract(*results*)

Parameters ***results*** [TODO] TODO

Returns TODO

set_next(*data*)

Parameters ***data*** [TODO] TODO

Returns `None`

45.2.3 DelayHRF

class DelayHRF(*input_hrf=<nipy.modalities.fmri.filters.Filter object at 0xb3957cc>, spectral=True, **keywords*)

Bases: `nipy.modalities.fmri.hrf.SpectralHRF`

Delay filter with spectral or Taylor series decomposition for estimating delays.

Liao et al. (2002).

__init__(*input_hrf=<nipy.modalities.fmri.filters.Filter object at 0xb3957cc>, spectral=True, **keywords*)

Parameters ***input_hrf*** [TODO] TODO

spectral [bool] TODO

keywords [dict] Passed through as keywords to the *hrf.SpectralHRF* constructor.

deltaPCA(*tmax=50.0, lower=-15.0, delta=None*)

Perform an expansion of fn, shifted over the values in delta. Effectively, a Taylor series approximation to fn(t+delta), in delta, with basis given by the filter elements. If fn is None, it assumes fn=IRF[0], that is the first filter.

Parameters ***tmax*** [float] TODO

lower [float] TODO

delta [[float]] TODO

Returns `None`

CHAPTER

FORTYSIX

MODALITIES.FMRI.FMRISTAT.INVERT

46.1 Module: modalities.fmri.fmristat.invert

invertR(*delta, IRF, niter=20*)

If IRF has 2 components (w0, w1) return an estimate of the inverse of r=w1/w0, as in Liao et al. (2002). Fits a simple arctan model to the ratio w1/w0.

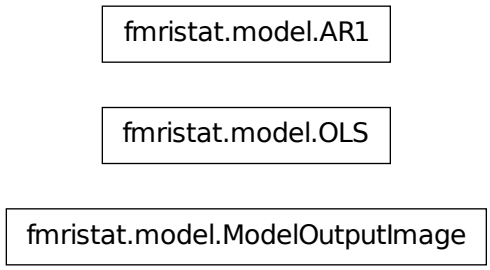
CHAPTER

FORTYSEVEN

MODALITIES.FMRI.FMRISTAT.MODEL

47.1 Module: modalities.fmri.fmristat.model

Inheritance diagram for nipy.modalities.fmri.fmristat.model:



This module defines the two default GLM passes of fmristat

47.2 Classes

47.2.1 AR1

class AR1 (*fmri_image, formula, rho, outputs=, [], volume_start_times=None*)

Second pass through fmri_image.

Parameters **fmri_image** : *FmriImageList*

formula : *nipy.modalities.fmri.protocol.Formula*

rho : Image of AR(1) coefficients.

__init__ (*fmri_image, formula, rho, outputs=, [], volume_start_times=None*)

execute()

47.2.2 ModelOutputImage

class ModelOutputImage (*filename, coordmap, shape, clobber=False*)
These images have their values filled in as the model is fit, and are saved to disk after being completely filled in.

They are saved to disk by calling the 'save' method.

The __getitem__ and __setitem__ calls are delegated to a private Image. An exception is raised if trying to get/set data after the data has been saved to disk.

__init__ (*filename, coordmap, shape, clobber=False*)

save ()
Save current Image data to disk as a .nii file.

47.2.3 OLS

class OLS (*fmri_image, formula, outputs=, [], volume_start_times=None*)
First pass through fmri_image.

Parameters **fmri_image** : *FmriImageList*

formula : *nipy.modalities.fmri.protocol.Formula*

__init__ (*fmri_image, formula, outputs=, [], volume_start_times=None*)

execute ()

47.3 Functions

estimateAR (*resid, design, order=1*)
Estimate AR parameters using bias correction from fMRIstat.

generate_output (*outputs, iterable, reshape=<function <lambda> at 0xb568bc4>*)
Write out results of a given output.

In the regression setting, results is generally going to be a scipy.stats.models.model.LikelihoodModelResults instance.

model_generator (*formula, data, volume_start_times, iterable=None, slicetimes=None, model_type=<class 'nipy.fixes.scipy.stats.models.regression.OLSModel'>, model_params=<function <lambda> at 0xb568454>*)
Generator for the models for a pass of fmristat analysis.

output_AR1 (*outfile, fmri_image, clobber=False*)
Create an output file of the AR1 parameter from the OLS pass of fmristat.

image: FmriImageList

output_F (*outfile, contrast, fmri_image, clobber=False*)

output_T (*outbase, contrast, fmri_image, effect=True, sd=True, t=True, clobber=False*)

Parameters **outbase** : string

Base filename that will be used to construct a set of files for the TContrast. For example, outbase='output.nii' will result in the following files (assuming defaults for all other params): output_effect.nii, output_sd.nii, output_t.nii

contrast : a TContrast

output_resid(*outfile, fmri_image, clobber=False*)
Create an output file of the residuals parameter from the OLS pass of fmristat.

Uses affine part of the first image to output resids unless fmri_image is an Image.

results_generator(*model_iterable*)
Generator for results from an iterator that returns (index, data, model) tuples.

See model_generator.

CHAPTER

FORTYEIGHT

MODALITIES.FMRI.FMRISTAT.NPZIMAGE

48.1 Module: modalities.fmri.fmristat.npzimage

Inheritance diagram for nipy.modalities.fmri.fmristat.npzimage:

A simple way of storing *nipy.core.image.image.Image*'s in .npz files.

Also includes NPZBuffer that keeps data in an ndarray instance until flushed to an .npz file.

At the time, NifTi file writing is broken – that's why NifTi files are not written.

48.2 Class

48.3 NPZBuffer

class NPZBuffer (*filename, grid, clobber=False*)

A temporary image that is saved to an npz Image when its flush method is called.

The attribute 'extra' should be a dictionary of arrays that will be included in the resuting .npz file when flush is called. This is a simple way to add extra information to Image files.

__init__ (*filename, grid, clobber=False*)

flush()

48.4 Functions

create_npz (*filename, grid, dtype=<type 'numpy.float32'>, clobber=False*)

Create an .npz Image, which consists of at least three arrays:

- data: the data array

- dimnames: the dimension names of the corresponding grid
- affine: the affine transformation of grid

load_npz (*filename*)

Load an .npz Image, this .npz file must have at least two arrays

- data: the data array
- dimnames: the dimension names of the corresponding grid
- affine: the affine transformation of grid

The remaining arrays of .npz file are stored as the 'extra' attribute of the Image.

save_npz (*filename, image, clobber=False, **extra*)

Save an .npz Image, which consists of at least three arrays:

- data: the data array
- dimnames: the dimension names of the corresponding grid
- affine: the affine transformation of grid

Image must have an affine transformation, which is the only part of the mapping that is saved.

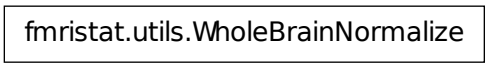
CHAPTER

FORTYNINE

MODALITIES.FMRI.FMRISTAT.UTILS

49.1 Module: modalities.fmri.fmristat.utils

Inheritance diagram for nipy.modalities.fmri.fmristat.utils:



Utilities for fmristat

49.2 WholeBrainNormalize

class WholeBrainNormalize (*fmri_image, mask=None*)

Bases: object

This class constructs a callable object that is sometimes used to normalize fMRI data before applying a GLM.

The normalization consists of dividing each point in the time series by the corresponding frame average of the fMRI image.

__init__ (*fmri_image, mask=None*)

Parameters *fmri_image* : A sequence of fMRI Images (can be a 4D Image) *mask* : A mask over which to average each fMRI volume.

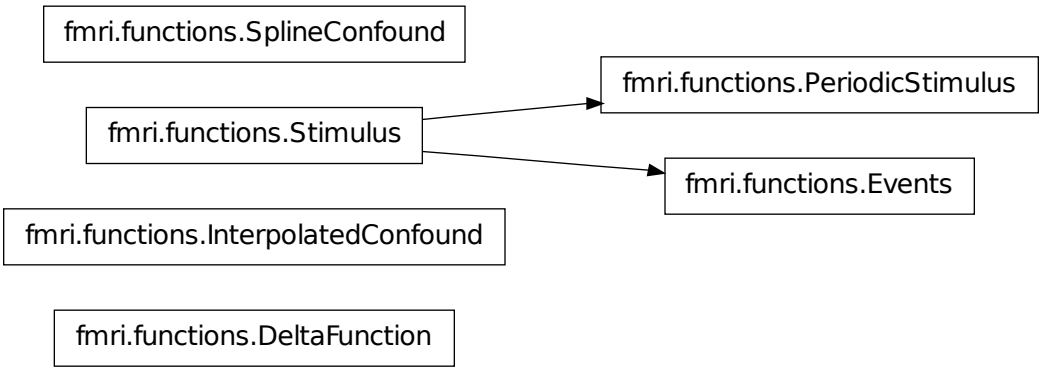
CHAPTER

FIFTY

MODALITIES.FMRI.FUNCTIONS

50.1 Module: `modalities.fmri.functions`

Inheritance diagram for `nipy.modalities.fmri.functions`:



This module defines some convenience functions of time.

Stimulus: a class to implement square-wave protocol based on a pair of sequences [times,values]

PeriodicStimulus: periodic Stimulus

Events: subclass of Stimulus to which events can be appended

DeltaFunction: an approximate delta function

SplineConfound: generate natural cubic splines with given knots

InterpolatedConfound: based on a sequence of [times, values], return a linearly interpolated confound

50.2 Classes

50.2.1 DeltaFunction

class DeltaFunction(*start=0.0, dt=0.02*)

A square wave approximate delta function returning 1/dt in interval [start, start+dt).

__init__(*start=0.0, dt=0.02*)

Parameters ***start*** [float] Beginning of delta function approximation.

dt [float] Width of delta function approximation.

50.2.2 Events

class Events(*name='stimulus', times=None, values=None*)

Bases: nipy.modalities.fmri.functions.Stimulus

TODO

__init__(*name='stimulus', times=None, values=None*)

Parameters ***fn*** [TODO] TODO

times [TODO] TODO

values [TODO] TODO

append(*start, duration, height=1.0*)

Append a square wave to an Event. No checking is made to ensure that there is no overlap with previously defined intervals – the assumption is that this new interval has empty intersection with all other previously defined intervals.

Parameters ***start*** [TODO] TODO

duration [TODO] TODO

height [float] TODO

Returns None

50.2.3 InterpolatedConfound

class InterpolatedConfound(*times=None, values=None, name='confound'*)

__init__(*times=None, values=None, name='confound'*)

Parameters

times : TODO TODO *values* : TODO TODO *name* : TODO TODO

50.2.4 PeriodicStimulus

class PeriodicStimulus(*n=1, start=0.0, duration=3.0, step=6.0, height=1.0, name='periodic stimulus'*)

Bases: nipy.modalities.fmri.functions.Stimulus

TODO

__init__ (*n=1, start=0.0, duration=3.0, step=6.0, height=1.0, name='periodic stimulus'*)

Parameters ***n*** [int] TODO

start [float] TODO

duration [float] TODO

step [float] TODO

height [float] TODO

50.2.5 SplineConfound

class SplineConfound (*df=4, knots=None, window=, [0, 1]*)
A natural spline confound with df degrees of freedom.

__init__ (*df=4, knots=None, window=, [0, 1]*)

Parameters ***df*** [int] TODO

knots [TODO] TODO

50.2.6 Stimulus

class Stimulus (*name='stimulus', times=None, values=None*)
TODO

__init__ (*name='stimulus', times=None, values=None*)

Parameters ***fn*** [TODO] TODO

times [TODO] TODO

values [TODO] TODO

50.3 Function

window (*f, r*)
Decorator to window a function between r[0] and r[1] (inclusive)

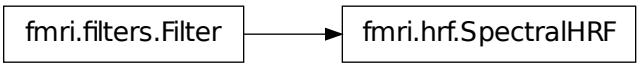
CHAPTER

FIFTYONE

MODALITIES.FMRI.HRF

51.1 Module: modalities.fmri.hrf

Inheritance diagram for nipy.modalities.fmri.hrf:



This module provides definitions of various hemodynamic response functions (hrf).

In particular, it provides Gary Glover's canonical HRF, AFNI's default HRF, and a spectral HRF.

51.2 SpectralHRF

class SpectralHRF(*input_hrf=<nipy.modalities.fmri.filters.Filter object at 0xb3957cc>, spectral=True, ncomp=2, names=, ['glover'], deriv=False, **keywords*)

Bases: nipy.modalities.fmri.filters.Filter

Delay filter with spectral or Taylor series decomposition for estimating delays.

Liao et al. (2002).

__init__(*input_hrf=<nipy.modalities.fmri.filters.Filter object at 0xb3957cc>, spectral=True, ncomp=2, names=, ['glover'], deriv=False, **keywords*)

Parameters ***input_hrf*** [TODO] TODO

spectral [bool] TODO

ncomp [int] TODO

names [TODO] TODO

deriv [bool] TODO

keywords [dict] passed as keyword arguments to *filters.Filter.__init__*

deltaPCA(*tmax=50.0, lower=-15.0, delta=None*)

Perform an expansion of fn, shifted over the values in delta. Effectively, a Taylor series approximation to fn(t+delta), in delta, with basis given by the filter elements. If fn is None, it assumes fn=IRF[0], that is the first filter.

```
>>> GUI = True
>>> import numpy as np
>>> from pylab import plot, title, show
>>> from nipy.modalities.fmri.hrf import glover, glover_deriv, SpectralHRF
>>>
>>> ddelta = 0.25
>>> delta = np.arange(-4.5,4.5+ddelta, ddelta)
>>> time = np.arange(0,20,0.2)
>>>
>>> hrf = SpectralHRF(glover)
>>>
>>> taylor = hrf.deltaPCA(delta=delta)
>>> curplot = plot(time, taylor.components[1](time))
>>> curplot = plot(time, taylor.components[0](time))
>>> curtitle=title('Shift using Taylor series -- components')
>>> show()
>>>
>>> curplot = plot(delta, taylor.coef[1](delta))
>>> curplot = plot(delta, taylor.coef[0](delta))
>>> curtitle = title('Shift using Taylor series -- coefficients')
>>> show()
>>>
>>> curplot = plot(delta, taylor.inverse(delta))
>>> curplot = plot(taylor.coef[1](delta) / taylor.coef[0](delta), delta)
>>> curtitle = title('Shift using Taylor series -- inverting w1/w0')
>>> show()
```

glover2GammaDENS(*peak_hrf, fwhm_hrf*)

Parameters ***peak_hfr*** [TODO] TODO

fwhm_hrf [TODO] TODO

Returns TODO

CHAPTER

FIFTYTWO

MODALITIES.FMRI.PCA

52.1 Module: modalities.fmri.pca

Inheritance diagram for nipy.modalities.fmri.pca:

This module provides a class for principal components analysis (PCA).

PCA is an orthonormal, linear transform (i.e., a rotation) that maps the data to a new coordinate system such that the maximal variability of the data lies on the first coordinate (or the first principal component), the second greatest variability is projected onto the second coordinate, and so on. The resulting data has unit covariance (i.e., it is decorrelated). This technique can be used to reduce the dimensionality of the data.

More specifically, the data is projected onto the eigenvectors of the covariance matrix.

52.2 PCA

class PCA(*image, tol=1.0000000000000001e-05, ext='.img', mask=None, pcatype='cor', design_keep=None, design_resid=None, **keywords*)

Bases: object

Compute the PCA of an image (over axis=0). Image coordmap should have a subcoordmap method.

__init__(*image, tol=1.0000000000000001e-05, ext='.img', mask=None, pcatype='cor', design_keep=None, design_resid=None, **keywords*)

Parameters ***image*** [*Image*] The image to be analysed

tol [float] TODO

ext [string] The file extension for the output image

mask [TODO] TODO

pcatype [string] TODO

design_resid [TODO] After projecting onto the column span of design_keep, data is projected off of the column span of this matrix.

design_keep [TODO] Data is projected onto the column span of design_keep.

fit()

Perform the computations needed for the PCA. This stores the covariance/correlation matrix of the data in the attribute 'C'. The components are stored as the attributes 'components', for an fMRI image these are the time series explaining the most variance.

Returns None

images(*which=, [0], output_base=None*)

Output the component images – by default, only output the first principal component.

Parameters ***which*** [TODO] TODO

output_base [TODO] TODO

Returns TODO

project(*Y, which='keep'*)

Parameters ***Y*** [TODO] TODO

which [string] TODO

Returns TODO

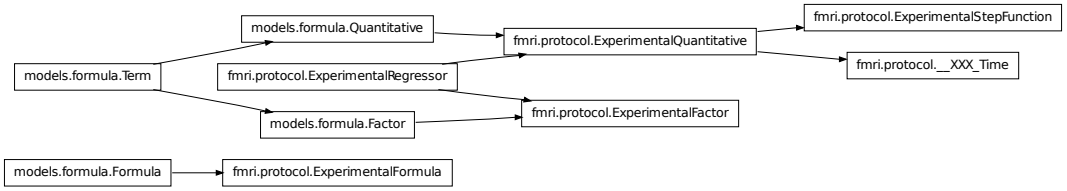
CHAPTER

FIFTYTHREE

MODALITIES.FMRI.PROTOCOL

53.1 Module: modalities.fmri.protocol

Inheritance diagram for nipy.modalities.fmri.protocol:



53.2 Classes

53.2.1 ExperimentalFactor

class ExperimentalFactor (*name, iterator, convolved=False, delta=True, dt=0.02*)

Bases: nipy.modalities.fmri.protocol.ExperimentalRegressor, nipy.fixes.scipy.stats.models.formula.Factor

Return a factor that is a function of experimental time based on an iterator. If the delta attribute is False, it is assumed that the iterator returns rows of the form:

type, start, stop

Here, type is a hashable object and start and stop are floats.

If delta is True, then the events are assumed to be delta functions and the rows are assumed to be of the form:

type, start

where the events are (square wave) approximations of a delta function, non zero on [start, start+dt).

Notes

self[key] returns the __UNCONVOLVED__ factor, even if the ExperimentalFactor has been convolved with an HRF.

__init__ (*name, iterator, convolved=False, delta=True, dt=0.02*)

Parameters **name** [TODO] TODO

iterator [TODO] TODO

convolved [bool] TODO

delta [bool] TODO

dt [float] TODO

fromiterator (*iterator, delimiter=', '*)

Determine an ExperimentalFactor from an iterator

Parameters **iterator** [TODO] TODO

delimiter [string] TODO

Returns `None`

main_effect ()

Return the 'main effect' for an ExperimentalFactor.

Returns *ExperimentalQuantitative*

names (*keep=False*)

Parameters **keep** [bool] TODO

Returns TODO

53.2.2 ExperimentalFormula

class ExperimentalFormula (*termlist, namespace={}*)

Bases: `nipy.fixes.scipy.stats.models.formula.Formula`

A formula with no intercept.

__init__ (*termlist, namespace={}*)

Create a formula from either: 1. a *formula* object

2. a sequence of *term* instances

3. one *term*

names (*keep=False*)

Parameters **keep** [bool] TODO

Returns TODO

53.2.3 ExperimentalQuantitative

class ExperimentalQuantitative (*name, fn, termname=None, **keywords*)

Bases: `nipy.modalities.fmri.protocol.ExperimentalRegressor`, `nipy.fixes.scipy.stats.models.formula.Quantitative`

Generate a regressor that is a function of time based on a function fn.

__init__(*name, fn, termname=None, **keywords*)

53.2.4 ExperimentalRegressor

class ExperimentalRegressor(*convolved=False, namespace={'time': <nipy.modalities.fmri.protocol.ExperimentalQuantitative object at 0x8aa8acc>}, termname='term'*)

Bases: object

__init__(*convolved=False, namespace={'time': <nipy.modalities.fmri.protocol.ExperimentalQuantitative object at 0x8aa8acc>}, termname='term'*)

Parameters ***convolved*** [bool] TODO

namespace [TODO] TODO

termname [string] TODO

convolve(*IRF*)

Parameters ***IRF*** [TODO] TODO

Returns self

convolved

names()

Returns TODO

53.2.5 ExperimentalStepFunction

class ExperimentalStepFunction(*name, iterator, **keywords*)

Bases: nipy.modalities.fmri.protocol.ExperimentalQuantitative

This returns a step function from an iterator returning tuples

(start, stop, height)

with height defaulting to 1 if not present.

__init__(*name, iterator, **keywords*)

Parameters **name** [TODO] TODO

iterator [TODO] TODO

keywords [dict] Passed through as the keywords to *ExperimentalQuantitative.__init__*

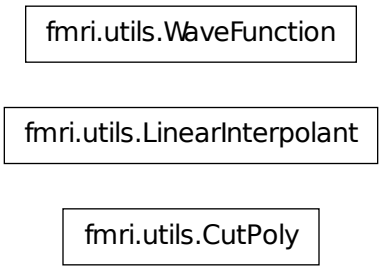
CHAPTER

FIFTYFOUR

MODALITIES.FMRI.UTILS

54.1 Module: modalities.fmri.utils

Inheritance diagram for nipy.modalities.fmri.utils:



54.2 Classes

54.2.1 CutPoly

class CutPoly (*power, trange=(None, None)*)

Bases: object

A polynomial function of the form f(t) = t^n with an optionally fixed time range on which the function exists.

__init__ (*power, trange=(None, None)*)

Paramters ***power*** [float] f(t) = t^power

trange [(float or None, float or None)] A tuple with the upper and lower bound of the function. None signifies no boundary. Default = (None, None)

54.2.2 LinearInterpolant

class LinearInterpolant (*x, y, fill_value=0.0*)
Bases: object

A little wrapper around scipy.interpolate call to force the interpolant to take a keywords argument 'time='.

__init__ (*x, y, fill_value=0.0*)

Parameters ***x*** [numpy.ndarray] A 1D array of monotonically increasing real values. x cannot include duplicate values (otherwise f is overspecified)

y [numpy.ndarray] An N-D array of real values. y's length along the interpolation axis must be equal to the length of x.

fill_value [float] If provided, then this value will be used to fill in for requested points outside of the data range.

Prerequisite len(x) == len(y)

54.2.3 WaveFunction

class WaveFunction (*start, duration, height*)
Bases: object

A square wave function of a specified start, duration and height. f(t) = height if (start <= t < start + duration), 0 otherwise

__init__ (*start, duration, height*)

Parameters ***start*** [float] The time of the rising edge of the square wave.

duration [float] The width of the square wave

height [float] The height of the square wave

54.3 Function

ConvolveFunctions (*fn1, fn2, interval, dt, padding_f=0.10000000000000001, normalize=(0, 0)*)
Convolve fn1 with fn2 – where fn1 may return a multidimensional output.

Parameters ***fn1*** [TODO] TODO

fn2 [TODO] TODO

interval [TODO] TODO

dt [TODO] TODO

padding_f [float] TODO

normalize [TODO] TODO

Returns TODO

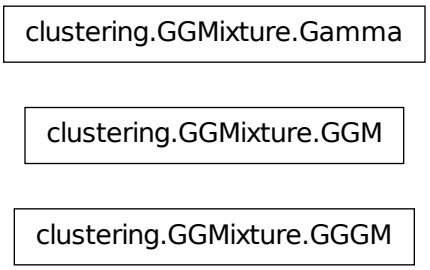
CHAPTER

FIFTYFIVE

NEUROSPIN.CLUSTERING.GGMIXTURE

55.1 Module: neurospin.clustering.GGMixture

Inheritance diagram for nipy.neurospin.clustering.GGMixture:



One-dimensional Gamma-Gaussian mixture density classes : Given a set of points the algo provides approcumate maximum likelihood estimates of the mixture distribution using an EM algorithm

Author: Bertrand Thirion and Merlin Keller 2005-2008

55.2 Classes

55.2.1 GGGM

class GGGM(*shape_n=1, scale_n=1, mean=0, var=1, shape_p=1, scale_p=1, mixt=array(, [0.33333333, 0.33333333, 0.33333333],)*)

This is the basic one dimensional Gamma-Gaussian-Gamma Mixture estimation class, where the frist gamma has a negative sign, while the second one has a positive sign 7 parameters are used: - shape_n: negative gamma shape - scale_n: negative gamma scale - mean: gaussian mean - var: gaussian variance - shape_p: positive gamma shape - scale_p: positive gamma scale - mixt: array of mixture parameter (weights of the n-gamma,gaussian and p-gamma)

__init__(*shape_n=1, scale_n=1, mean=0, var=1, shape_p=1, scale_p=1, mixt=array(, [0.33333333, 0.33333333, 0.33333333],)*)

Estep (*x) update probabilistic memberships of the three components x: input data, shape=(nbitems) z: probabilisticmembership, shape =(nbitems, 3*)

Mstep (*x, z) Update the parameters of the three components through ML approach x: input data, shape=(nbitems) z: probabilisticmembership, shape =(nbitems, 3*)

ROC (*x*)
ROC curve for seperating positive Gamma distribution from two other modes, predicted by current parameter values -x: vector of observations

Output: P P[0]: False positive rates P[1]: True positive rates

SAEM (*x, burnin=100, niter=200, verbose=False*)
SAEM estimation procedure: Input: -x: vector of observations -burnin: number of burn-in SEM iterations (discarded values) -niter: maximum number of SAEM iterations for convergence to local maximum -verbose (0/1): verbosity level Gaussian disribution mean is fixed to zero Output: -LL: successive log-likelihood values

check (*x*)

component_likelihood (*x*)
ng,y,pg = self.component_likelihood(x) Compute the likelihood of the data x under the three components negative gamma, gaussina, positive gaussian

estimate (*x, niter=100, delta=0.0001, bias=0, verbose=0*)
Whole EM estimation procedure: x : input data, should be an array of size nbitems niter = 100 : max number of iterations delta = 1.e-4: increment in LL at xhich convergence is declared bias=0 : possible hard constrain on the gaussian variance (to avoid shrinkage); when bias=0, no contrain is applied verbose=1: verbosity level

init (*x, mixt=array(, [1., 1., 1.],)*)
initialization of the differnt parameters - mixt = np.ones(3,'d') gives the prior mixture parameters the other parameters are initialized using standard approaches

init_fdr (*x, dof=-1*)
Initilization of the class based on a fdr heuristic: the probability to be in the positive component is proportional to the 'positive fdr' of the data. The same holds for the negative part. The point is that the gamma parts should model nothing more that the tails of the distribution

parameters ()

posterior (*x*)
ng,y,pg = self.posterior(x) Compute the posterior probability of observing the data x under the three components negative gamma, gaussina, positive gaussian

show (*x, figname=None*)
vizualization of the MM, based on the empirical histogram of x

55.2.2 GGM

class GGM (*shape=1, scale=1, mean=0, var=1, mixt=0.5*)
This is the basic one dimensional Gaussian-Gamma Mixture estimation class Note that it can work with positive or negative values, as long as there is at least one positive value. NB : The gamma distribution is defined only on positive values. 5 parameters are used: - mean: gaussian mean - var: gaussian variance - shape: gamma shape - scale: gamma scale - mixt: mixture parameter (weight of the gamma)

__init__ (*shape=1, scale=1, mean=0, var=1, mixt=0.5*)

Estep (*x) Estimation of the likelihood of the data for the three compartments x = inpute data (shape = (nbitems)) z = likelihoods (shape=(nbitems, 2)*)

Mstep(*x, z*)
self.Mstep(x,z) Estimation of the parameters of the model x = the data z = probabilistic membership matrix

check(*x*)

estimate(*x, niter=100, delta=0.0001, verbose=False*)
Complete EM estimation procedure x : data (shape = nbitems) niter =100: max nb of iterations delta = 0.001: criterion for convergence

parameters()

posterior(*x*)
ng,y,pg = self.posterior(x) Compute the posterior probability of observing the data x under the three components negative gamma, gaussina, positive gaussian

show(*x*)
vizualisation of the mm based on the empirical histogram of x

55.2.3 Gamma

class Gamma(*shape=1, scale=1*)
This is the basic one dimensional Gaussian-Gamma Mixture estimation class Note that it can work with positive or negative values, as long as there is at least one positive value. NB : The gamma distribution is defined only on positive values. 5 parameters are used: - mean: gaussian mean - var: gaussian variance - shape: gamma shape - scale: gamma scale - mixt: mixture parameter (weight of the gamma)

__init__(*shape=1, scale=1*)

check(*x*)

estimate(*x, eps=9.9999999999999995e-08*)
ML estimation of the Gamma parameters

parameters()

55.3 Functions

compute_c(*x, z, eps=1.0000000000000001e-05*)
this function returns the mle of the shape parameter if a 1D gamma density

dichopsi_log(*u, v, y, eps=1.0000000000000001e-05*)
this function implements the dichotomic part of the solution of the psi(c)-log(c)=y

psi_solve(*y, eps=1.0000000000000001e-05*)
This function solves solve psi(c)-log(c)=y by dichotomy

test_GGGM()

test_GGM()

test_Gamma_parameters()

CHAPTER

FIFTYSIX

NEUROSPIN.CLUSTERING.BOOTSTRAP_HC

56.1 Module: neurospin.clustering.bootstrap_hc

This module provides some code to perform bootstrap of Ward's hierarchical clustering This is useful to statistically validate clustering results. theory see:

Author : Bertrand Thirion, 2008

56.2 Functions

demo_ward_msb (*n=30, d=30, niter=1000*)

basic demo for the ward_msb procedure in that case the dominant split with 2 clusters should have dominant p-val INPUT: - n,d : the dimensions of the dataset -niter : the number of bootrstraps

ward_msb (*X, niter=1000*)

multi-scale bootstrap procedure INPUT: - X array of shape (n,p) where n is the number of items to be clustered p is their dimensions - niter=1000 number of iterations of the bootstrap OUPUT: - t the resulting tree clustering the associated subtrees is defined as t.list_of_subtrees() there are precisely n such subtrees - cpval: array of shape (n) : the corrected p-value of the clusters - upval: array of shape (n) : the uncorrected p-value of the clusters

CHAPTER

FIFTYSEVEN

NEUROSPIN.CLUSTERING.CLUSTERING

57.1 Module: neurospin.clustering.clustering

Clustering routines. Author: Bertrand Thirion (INRIA Saclay, Orsay, France), 2004-2008.

kmeans (*X, nbclusters=2, Labels=None, maxiter=300, delta=0.0001, verbose=0*)

Centers, Labels, J = Cmeans(X,nbclusters,Labels,maxiter,delta) cmeans clustering algorithm

INPUT : - A data array X, supposed to be written as (n*p)

where n = number of features, p =number of dimensions

- nbclusters (int), the number of desired clusters
- Labels=None n array of predefined Labels. if None or inadequate a random initilization is performed.
- maxiter(int, =300 by default), the maximum number of iterations before convergence
- delta(double, =0.0001 by default),

the relative increment in the results before declaring convergence - verbose=0: verboseity mode

OUPUT : - Centers: array of size nbclusters*p, the centroids of

the resulting clusters

- Labels : arroy of size n, the discrete labels of the input items;
- J the final value of the criterion

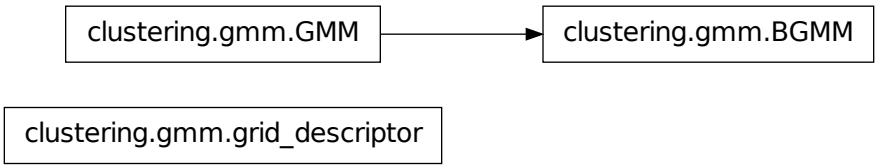
CHAPTER

FIFTYEIGHT

NEUROSPIN.CLUSTERING.GMM

58.1 Module: neurospin.clustering.gmm

Inheritance diagram for nipy.neurospin.clustering.gmm:



Gaussian Mixture Model Class: contains the basic fields and methods of GMMs the high level functions are/should be binded in C

Author : Bertrand Thirion, 2006-2009

58.2 Classes

58.2.1 BGMM

class BGMM(*k=1, dim=1, prec_type=1, centers=None, precision=None, weights=None*)

Bases: nipy.neurospin.clustering.gmm.GMM

This class implements Bayesian diagonal GMMs (prec_type = 1) Besides the standard fiels of GMMs, this class contains the follwing fields - prior_centers : array of shape (k,dim): the prior on the components means - prior_precision : array of shape (k,dim): the prior on the components precisions - prior_dof : array of shape (k): the prior on the dof (should be at least equal to dim) - prior_mean_scale : array of shape (k): scaling factor of the prior precision on the mean - prior_weights : array of shape (k) the prior on the components weights - mean_scale : array of shape (k): scaling factor of the posterior precision on the mean - dof : array of shape (k): the posterior dofs

__init__(*k=1, dim=1, prec_type=1, centers=None, precision=None, weights=None*)

Gibbs_estimate(*X, niter=1000, method=1*)

Estimation of the BGMM using Gibbs sampling INPUT: - X array of shape (nbitems,dim) the input data

- niter = 1000, the maximal number of iterations of the Gibbs sampling - method = 1: boolean to state whether covariance are fixed (0 ; normal model) or variable (1 ; normal-wishart model) OUTPUT: - label: array of shape nbitems: resulting MAP labelling

Gibbs_estimate_and_sample(*X, niter=1000, method=1, gd=None, nsamp=1000, verbose=0*)
Estimation of the BGMM using Gibbs sampling and sampling of the posterior on test points INPUT: - X array of shape (nbitems,dim) the input data - niter = 1000, the maximal number of iterations of the Gibbs sampling - method = 1: boolean to state whether covariance are fixed (0 ; normal model) or variable (1 ; normal-wishart model) - gd = None, a grid descriptor, i.e. the grid on chich the model is sampled if gd==None, X is used as Grid - nsamp = 1000 number of draws of the posterior -verbose = 0: the verboseity level OUTPUT: - Li : array of shape (nbnodes): the average log-posterior - label: array of shape (nbitems): resulting MAP labelling

VB_estimate(*X, niter=100, delta=0.0001*)
Estimation of the BGMM using a Variational Bayes approach INPUT: - X array of shape (nbitems,dim) the input data - niter = 100, the maximal number of iterations of the VB algo - delta = 0.0001, the increment in log-likelihood to declare convergence OUTPUT: - label: array of shape nbitems: resulting MAP labelling

VB_estimate_and_sample(*X, niter=1000, delta=0.0001, gd=None, verbose=0*)
Estimation of the BGMM using a Variational Bayes approach, and sampling of the model on test points in order to have an estimate of the posterior on these points INPUT: - X array of shape (nbitems,dim) the input data - niter = 100, the maximal number of iterations of the VB algo - delta = 0.0001, the increment in log-likelihood to declare convergence - gd = None a grid descriptor, i.e. the grid on chich the model is sampled if gd==None, X is used as Grid - verbose = 0: the verbosity mode OUTPUT: - Li : array of shape (nbnodes): the average log-posterior - label: array of shape nbitems: resulting MAP labelling

VB_sample(*gd, X=None*)
Sampling of the BGMM model on test points (the 'grid')in order to have an estimate of the posterior on these points INPUT: - gd = a grid descriptor, i.e. the grid on chich the BGMM is sampled - X = None: used for plotting (empirical data) OUTPUT: - Li : array of shape (nbnodes,self.k): the posterior for each node and component

check_priors()
Check that the meain fields have correct dimensions

sample_on_data(*grid*)
Sampling of the BGMM model on test points (the 'grid')in order to have an estimate of the posterior on these points INPUT: - grid: a set of points from which the posterior should be smapled OUTPUT: - Li : array of shape (nbnodes,self.k): the posterior for each node and component

set_empirical_priors(*X*)
Set the prior in a natural (almost uninformative) fashion given a dataset X INPUT: - the BGMM priors

set_priors(*prior_centers=None, prior_weights=None, prior_precision=None, prior_dof=None, prior_mean_scale=None*)
Set the prior of the BGMM

58.2.2 GMM

class GMM(*k=1, dim=1, prec_type=1, centers=None, precision=None, weights=None*)
This is the basic GMM class GMM.k is the number of components in the mixture GMM.dim is the dimension of the data GMM.centers is an array that contains all the centers of the components

shape (GMM.k,GMM.dim)

GMM.precision is an array that contains all the precision of the components its shape varies according to GMM.prec_type

GMM.prec_type type of the precision matrix - O: full coavriance matrix, one for each component. shape = (GMM.k,GMM.dim**2) - 1 : diagonal covariance matrix, one for each components. shape = (GMM.k,GMM.dim) - 2 : diagonal covariance matrix, the same for all component. shape = (1,GMM.dim) GMM.weights contains the weights of the components in the mixture GMM.estimated is a binary variable that indicates whether the model has been instantiated or not

__init__ (*k=1, dim=1, prec_type=1, centers=None, precision=None, weights=None*)

BIC (*LL, n*)
Computing the value of the BIC critrion of the current GMM, given its average log-likelihood LL

assess_divergence ()

check ()
Checking the shape of sifferent matrices involved in the model

check_data (*data*)
Checking that the data is in correct format

estimate (*data, Labels=None, maxiter=300, delta=0.001, ninit=1*)
Estimation of the GMM based on data and an EM algorithm INPUT data : (n*p) feature array, n = nb items, p=feature dimension Labels=None : prior labelling of the data (this may improve convergence) maxiter=300 : max number of iterations of the EM algorithm delta = 0.001 : criterion on the log-likelihood increments to declare converegence ninit=1 : number of possible iterations of the GMM estimation OUTPUT Labels : (n) array of type ('i') discrete labelling of the data items into clusters LL : (float) average log-likelihood of the data BIC : (float) associated BIC criterion

optimize_with_BIC (*data, kvals=None, maxiter=300, delta=0.001, ninit=1, verbose=0*)
Find the optimal GMM using BIC criterion. The method is run with all the values in kmax for k INPUT data : (n*p) feature array, n = nb items, p=feature dimension kvals=None : range of values for k. maxiter=300 : max number of iterations of the EM algorithm delta = 0.001 : criterion on the log-likelihood increments to declare converegence ninit=1 : number of possible iterations of the GMM estimation verbsose=0: verbosity mode OUTPUT Labels : (n) array of type ('i') discrete labelling of the data items into clusters LL : (float) average log-likelihood of the data BIC : (float) associated BIC criterion

partition (*data*)
Partitioning the data according to the gmm model INPUT data : (n*p) feature array, n = nb items, p=feature dimension OUTPUT Labels : (n) array of type ('i') discrete labelling of the data items into clusters LL : (n) array of type ('d') log-likelihood of the data

sample (*gd, X, verbose=0*)
Evaluating the GMM on some new data INPUT data : (n*p) feature array, n = nb items, p=feature dimension OUTPUT Labels : (n) array of type ('i') discrete labelling of the data items into clusters LL : (n) array of type ('d') log-likelihood of the data

set_k (*k*)
To set the value of k

show (*X, gd, density=None, nbf=-1*)
Function to plot a GMM -WIP Currently, works only in 1D and 2D

show_components (*X, gd, density=None, nbf=-1*)
Function to plot a GMM -WIP Currently, works only in 1D and 2D

test (*data*)
Evaluating the GMM on some new data INPUT data : (n*p) feature array, n = nb items, p=feature dimension OUTPUT

LL : (n) array of type ('d') log-likelihood of the data

58.2.3 grid_descriptor

class **grid_descriptor**(*dim=1*)

A tiny class to handle cartesian grids

__init__(*dim=1*)

getinfo(*lim, nbs*)

make_grid()

CHAPTER

FIFTYNINE

NEUROSPIN.CLUSTERING.HIERARCHICAL_C

59.1 Module: neurospin.clustering.hierarchical_clustering

Inheritance diagram for nipy.neurospin.clustering.hierarchical_clustering:

These routines perform some hierrachical agglomerative clustering of some input data. The following alternatives are proposed: - Distance based average-link - Similarity-based average-link - Distance based maximum-link - Ward's algorithm under graph constraints - Ward's algorithm without graph constraints

In this latest version, the results are returned in a 'WeightedForest' structure, which gives access to the clustering hierarchy, facilitates the plot of the result etc.

For back-compatibility, *_segment versions of the algorithms have been appended, with the old API (except the qmax parameter, which now represents the number of wanted clusters)

Author : Bertrand Thirion,Pamela Guevara, 2006-2009

59.2 Class

59.3 WeightedForest

class WeightedForest (*V, parents=None, height=None*)

Bases: nipy.neurospin.graph.graph.Forest

This is a weighted Forest structure, i.e. a tree - ecah node has one parent and children (hierarchical structure) - some of the nodes can be viewed as leaves, other as roots - the edges within a tree are associated with a weight: +1 from child to parent -1 from parent to child - additionally, the nodes have a value, which is called 'height', especially useful from dendrograms

fields: - V : (int,>0) the number of vertices - E : (int) the number of edges - parents: array of shape (self.V) the parent array - edges: array of shape (self.E,2) reprensenting pairwise neighbors - weights, array of shape (self.E), +1/-1 for scending/descending links - children: list of arrays that represents the childs of any node - height: array of shape(self.V)

__init__ (*V, parents=None, height=None*)
INPUT: V: the number of edges of the graph parents = None: array of shape (V) the parents of the graph by default, the parents are set to range(V), i.e. each node is its own parent, and each node is a tree height=None: array of shape(V) the height of the nodes

check_compatible_height ()
Check that height[parents[i]]>=height[i] for all nodes

fancy_plot (*validleaves*)
Idem plot, but the valid edges are enahanced

fancy_plot_ (*valid*)
Idem plot, but the valid edges are enahanced

get_height ()
get the height array

list_of_subtrees ()
returns the list of all non-trivial subtrees in the graph Caveat: theis function assumes that the vertices are sorted in a way such that parent[i]>i forall i Only the leaves are listeed, not the subtrees themselves

partition (*threshold*)
partition the tree according to a cut criterion

plot ()
Plot the dendrogram associated with self the rank of the data in the dendogram is returned

plot2 (*addNodes=False, font_size=10, cl_size=None*)
Plot the dendrogram associated with self

plot_height ()
plot the height of the non-leaves nodes

set_height (*height=None*)
set the height array

split (*k*)
idem as partition, but a number of components are supplied instead

59.4 Functions

Average_Link_Distance (*D, verbose=0*)
Average link clustering based on a pairwise distance matrix.

Average_Link_Distance(D,verbose=0): INPUT: - D: a (n,n) distance matrix between some items - verbose=0, verbosity level OUTPUT: -t a weightForest structure that represents the dendrogram of the data NOTE: this method has not been optimized

Average_Link_Distance_segment (*D, stop=-1, qmax=1, verbose=0*)
Average link clustering based on a pairwise distance matrix.

Average_Link_Distance(D,stop=-1,qmax=-1,verbose=0): INPUT: - D: a (n,n) distance matrix between some items - stop=-1: stopping criterion, i.e. distance threshold at which further merges are forbidden By default, all merges are performed - qmax = 1; the number of desired clusters (in the limit of stop) - verbose=0, verbosity level OUTPUT: -u: a labelling of the graph vertices according to the criterion -cost the cost of each merge step during the clustering procedure NOTE: this method has not been optimized

Average_Link_Euclidian (*X, verbose=0*)
Average link clustering based on data matrix. INPUT: - X: a (nbitem,dim) array from which an Euclidian

distance matrix is computed - verbose=0, verbosity level OUTPUT: -t a weightForest structure that represents the dendrogram of the data NOTE: this method has not been optimized

Average_Link_Graph(*G*)
Agglomerative function based on a (hopefully sparse) similarity graph INPUT: - G the input graph OUPUT: - t a weightForest structure that represents the dendrogram of the data CAVEAT: In that case, the homogeneity is associated with high similarity (as opposed to low cost as in most clustering procedures, e.g. distance-based procedures). Thus the tree is created with negated affinity values, in roder to respect the traditional ordering of cluster potentials. individual points have the potential (-np.infty). This problem is handled transparently inthe associated segment functionp.

Average_Link_Graph_segment(*G, stop=0, qmax=1, verbose=0*)
Agglomerative function based on a (hopefully sparse) similarity graph INPUT: - G the input graph - stop = 0: the stopping crterion - qmax=1: the number of desired clusters (in the limit of the stopping criterion) OUPUT: -u: a labelling of the graph vertices according to the criterion -cost the cost of each merge step during the clustering procedure

Maximum_Link_Distance(*D, stop=-1, qmax=-1, verbose=0*)
maximum link clustering based on a pairwise distance matrix. Maximum_Link_Distance(D,stop=0,qmax=-1,verbose=0): INPUT: - D: a (n,n) distance matrix between some items - verbose=0, verbosity level OUTPUT: -t a weightForest structure that represents the dendrogram of the data NOTE: this method has not been optimized

Maximum_Link_Distance_segment(*D, stop=-1, qmax=1, verbose=0*)
maximum link clustering based on a pairwise distance matrix. Maximum_Link_Distance(D,stop=0,qmax=-1,verbose=0): INPUT: - D: a (n,n) distance matrix between some items - qmax = 1: the number of desired clusters (in the limit of stop) - stop=-1: stopping criterion, i.e. distance threshold at which further merges are forbidden By default (stop=-1), all merges are performed - verbose=0, verbosity level OUTPUT: -u: a labelling of the graph vertices according to the criterion -cost the cost of each merge step during the clustering procedure NOTE: this method has not been optimized

Maximum_Link_Euclidian(*X, verbose=0*)
Maximum link clustering based on data matrix. INPUT: - X: a (nbitem,dim) array from which an Euclidian distance matrix is computed - verbose=0, verbosity level OUTPUT: -t a weightForest structure that represents the dendrogram of the data NOTE: this method has not been optimized

Ward(*G, feature, verbose=0*)
Agglomerative function based on a topology-defining graph and a feature matrix. INPUT: - G the input graph (a topological graph essentially) - feature (G.V,dim_feature) array that yields some vectorial information related to the graph vertices OUPUT: -t: a WeightedForest structure that represents the dendrogram NOTE: When G has more than 1 connected component, t is no longer a tree. This case is handled cleanly now

Ward_quick(*G, feature, verbose=0*)
Agglomerative function based on a topology-defining graph and a feature matrix. INPUT: - G the input graph (a topological graph essentially) - feature (G.V,dim_feature) array that yields some vectorial information related to the graph vertices OUPUT: - t a weightForest structure that represents the dendrogram of the data NOTE: - Hopefully a quicker version - A euclidean distance is used in the feature space CAVEAT : only approximate

Ward_quick_segment(*G, feature, stop=-1, qmax=1, verbose=0*)
Agglomerative function based on a topology-defining graph and a feature matrix. INPUT: - G the input graph (a topological graph essentially) - feature (G.V,dim_feature) array that yields some vectorial information related to the graph vertices - stop = -1: the stopping crterion if stop==-1, then no stopping criterion is used - qmax=1: the maximum number of desired clusters (in the limit of the stopping criterion) OUPUT: -u: a labelling of the graph vertices according to the criterion -cost the cost of each merge step during the clustering procedure NOTE: - Hopefully a quicker version - A euclidean distance is used in the feature space CAVEAT : only approximate

Ward_segment(*G, feature, stop=-1, qmax=1, verbose=0*)
Agglomerative function based on a topology-defining graph and a feature matrix. INPUT: - G the input graph (a topological graph essentially) - feature (G.V,dim_feature) array that yields some vectorial information related to

the graph vertices - stop = -1: the stopping crterion if stop==-1, then no stopping criterion is used - qmax=1: the maximum number of desired clusters (in the limit of the stopping criterion) OUPUT: -u: a labelling of the graph vertices according to the criterion -cost the cost of each merge step during the clustering procedure NOTE: - A euclidean distance is used in the feature space - caveat : when the number of cc in G (nbcc) is greter than qmax, u contains nbcc values, not qmax !

Ward_simple (*feature, verbose=0, quick=1*)
Ward clustering based on a Feature matrix

t = Ward(feature,verbose=0): INPUT: - feature: a (n,p) feature matrix between some items representing n p-dimenional items to be clustered - verbose=0, verbosity level OUTPUT: -t a weightForest structure that represents the dendrogram of the data NOTE: this method uses the optimized C routine if "quick" is true.

fusion (*K, pop, i, j, k) modifies the graph K to merge nodes i and j into nodes k the similarity values are weighted averaged, where pop, [i], and pop, [j], yield the relative weights. this is used in average_link_slow (deprecated*)

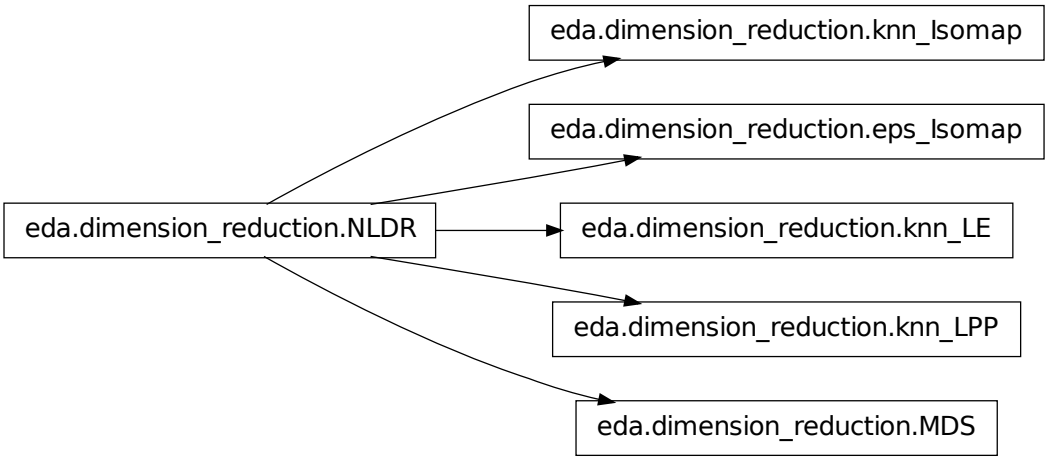
CHAPTER

SIXTY

NEUROSPIN.EDA.DIMENSION_REDUCTION

60.1 Module: neurospin.eda.dimension_reduction

Inheritance diagram for nipy.neurospin.eda.dimension_reduction:



This modules contains several classes to perform non-linear dimension reduction. Each class has 2 methods, 'train' and 'test' - 'train' performs the computation of low-simensional data embedding and the information to generalize to new data - 'test' computes the embedding for new dsamples of data This is done for - Multi-dimensional scaling - Isompap (knn or eps-neighb implementation) - Locality Preseving projections (LPP) - laplacian embedding (train only)

Future developpements will include some supervised cases, e.g. LDA,LDE and the estimation of the latent dimension, at least in simple cases.

60.2 Classes

60.2.1 MDS

class MDS (*X=None, rdim=1, fdim=1*)

Bases: nipy.neurospin.eda.dimension_reduction.NLDR

This is a particular class that perfoms linear dimension reduction using multi-dimensional scaling besides the fields of NDLR, it contains the following ones: - trained: trained==1 means that the system has been trained and can generalize - embedding: array of shape (nbitems,rdim) this is representation of the training data - offset: array of shape(nbitems) affine part of the embedding - projector: array of shape(fdim,rdim) linear part of the embedding

__init__ (*X=None, rdim=1, fdim=1*)

test (*X*)

chart = MDS.test(X,verbose=0) X = array of shape(nbitems,fdim) new data points to be embedded verbose=0 : verbosity mode chart: resulting rdim-dimensional represntation

train (*verbose=0*)

chart = MDS.train(verbose=0) verbose=0 : verbosity mode chart: resulting rdim-dimensional represntation

60.2.2 NLDR

class NLDR (*X=None, rdim=1, fdim=1*)

This is a generic class for dimension reduction techniques the main fields are - train_data : the input dataset from which the DR is perfomed - fdim=1 - rdim=1

__init__ (*X=None, rdim=1, fdim=1*)

check_data (*X*)

set_train_data (*X*)

test (*X*)

train ()

60.2.3 eps_Isomap

class eps_Isomap (*X=None, rdim=1, fdim=1*)

Bases: nipy.neurospin.eda.dimension_reduction.NLDR

This is a particular class that perfoms linear dimension reduction using eps-ball neighbor modelling and isomapping. besides the fields of NDLR, it contains the following ones: - eps : eps-ball model used in the knn graph building - G : resulting graph based on the training data - trained: trained==1 means that the system has been trained and can generalize - embedding: array of shape (nbitems,rdim) this is representation of the training data - offset: array of shape(nbitems) affine part of the embedding - projector: array of shape(fdim,rdim) linear part of the embedding

__init__ (*X=None, rdim=1, fdim=1*)

test (*X*)

chart = eps_Isomap.test(X,verbose=0) INPUT X = array of shape(nbitems,fdim) new data points to be embedded verbose=0 : verbosity mode OUTPUT chart: resulting rdim-dimensional represntation

train (*eps=1.0, p=300, verbose=0*)
chart = eps_Isomap.train(X,verbose=0) INPUT eps= 1.0: self.eps p = 300 number points used in the low dimensional approximation - verbose=0 : verbosity mode OUTPUT: - chart = eps_Isomap.embedding

60.2.4 knn_Isomap

class knn_Isomap (*X=None, rdim=1, fdim=1*)
Bases: nipy.neurospin.eda.dimension_reduction.NLDR

This is a particular class that perfoms linear dimension reduction using k nearest neighbor modelling and isomapping. besides the fields of NDLR, it contains the following ones: - k : number of neighbors in the knn graph building - G : resulting graph based on the training data - trained: trained==1 means that the system has been trained and can generalize - embedding: array of shape (nbitems,rdim) this is representation of the training data - offset: array of shape(nbitems) affine part of the embedding - projector: array of shape(fdim,rdim) linear part of the embedding

__init__ (*X=None, rdim=1, fdim=1*)

test (*X*)
chart = knn_Isomap.test(X,verbose=0) INPUT X = array of shape(nbitems,fdim) new data points to be embedded verbose=0 : verbosity mode OUTPUT chart: resulting rdim-dimensional represntation

train (*k=1, p=300, verbose=0*)
chart = knn_Isomap.train(verbose=0) INPUT: - k=1 : k in the knn system - p=300 : number points used in the low dimensional approximation - verbose=0 : verbosity mode OUTPUT: - chart = knn_Isomap.embedding

60.2.5 knn_LE

class knn_LE (*X=None, rdim=1, fdim=1*)
Bases: nipy.neurospin.eda.dimension_reduction.NLDR

This is a particular class that perfoms linear dimension reduction using k nearest neighbor modelling and laplacian embedding. besides the fields of NDLR, it contains the following ones: - k : number of neighbors in the knn graph building - G : resulting graph based on the training data - trained: trained==1 means that the system has been trained and can generalize - embedding: array of shape (nbitems,rdim) this is representation of the training data NB: to date, only the training part (embedding computation) is considered

__init__ (*X=None, rdim=1, fdim=1*)

train (*k=1, verbose=0*)
chart = knn_LE.train(k=1,verbose=0) INPUT: - k=1 : k in the knn system - verbose=0 : verbosity mode OUTPUT: - chart = knn_LE.embedding

60.2.6 knn_LPP

class knn_LPP (*X=None, rdim=1, fdim=1*)
Bases: nipy.neurospin.eda.dimension_reduction.NLDR

This is a particular class that perfoms linear dimension reduction using k nearest neighbor modelling and locality preserving projection (LPP). besides the fields of NDLR, it contains the following ones: - k : number of neighbors in the knn graph building - G : resulting graph based on the training data - trained: trained==1 means that the system has been trained and can generalize - embedding: array of shape (nbitems,rdim) this is representation of the training data - projector: array of shape(fdim,rdim) linear part of the embedding

__init__ (*X=None, rdim=1, fdim=1*)

test (*X*)
chart = knn_LPP.test(X,verbose=0) INPUT X = array of shape(nbitems,fdim) new data points to be embedded verbose=0 : verbosity mode OUTPUT chart: resulting rdim-dimensional represntation

train (*k=1, verbose=0*)
chart = knn_LPP.train(verbose=0) INPUT: - k=1 : k in the knn system - verbose=0 : verbosity mode OUTPUT: - chart = knn_LPP.embedding

60.3 Functions

CCA (*X, Y, eps=9.9999999999999998e-13*)
Canonical corelation analysis of two matrices INPUT: - X and Y are (nbitem,p) and (nbitem,q) arrays that are analysed - eps=1.e-12 is a small biasing constant to grant invertibility of the matrices OUTPUT - ccs: the canconical correlations NOTE - It is expected that nbitem>>max(p,q) - In general it makes more sense if p=q

Euclidian_distance (*X, Y=None*)
Considering the rows of X (and Y=X) as vectors, compute the distance matrix between each pair of vector

Euclidian_mds (*X, dim, verbose=0*)
returns a dim-dimensional MDS representation of the rows of X using an Euclidian metric

LE (*G, dim, verbose=0, maxiter=1000*)
Laplacian Embedding of the data returns the dim-dimensional LE of the graph G chart = LE(G,dim,verbose=0,maxiter=1000) INPUT: - G : Weighted graph that represents the data - dim=1 : number of dimensions - verbose = 0: verbosity level - maxiter=1000: maximum number of iterations of the algorithm OUTPUT - chart, array of shape(G.V,dim) NOTE : In fact the current implementation retruns what is now referred to a diffusion map at time t=1

LE_dev (*G, dim, verbose=0, maxiter=1000*)
Laplacian Embedding of the data returns the dim-dimensional LE of the graph G INPUT: - G : Weighted graph that represents the data - dim=1 : number of dimensions - verbose = 0: verbosity level - maxiter=1000: maximum number of iterations of the algorithm OUTPUT - chart, array of shape(G.V,dim)

LPP (*G, X, dim, verbose=0, maxiter=1000) INPUT: - G : Weighted graph that represents the data - X : related input dataset - dim=1 : number of dimensions - verbose = 0: verbosity level - maxiter=1000: maximum number of iterations of the algorithm OUTPUT -proj, array of shape(X.shape, [1], dim*)

Orthonormalize (*M) orthonormalize the columns of M (Gram-Schmidt procedure*)

check_isometry (*G, chart, nseeds=100, verbose=0*)
A simple check of the Isometry: look whether the output distance match the intput distances for nseeds points OUTPUT: - a proportion factor to optimize the metric

infer_latent_dim (*X, verbose=0, maxr=-1*)
r = infer_latent_dim(X,verbose = 0) Infer the latent dimension of an aray assuming data+gaussian noise mixture INPUT: - an array X - verbose=0 : verbositry level - maxr=-1 maximum dimension that can be achieved if maxr = -1, this is equal to rank(X) OUPTUT - r the inferred dimension

isomap (*G, dim=1, p=300, verbose = 0) Isomapping of the data return the dim-dimensional ISOMAP chart that best represents the graph G INPUT: - G : Weighted graph that represents the data - dim=1 : number of dimensions - p=300 : nystrom reduction of the problem - verbose = 0: verbosity level OUTPUT - chart, array of shape(G.V, dim*)

isomap_dev (*G, dim=1, p=300, verbose=0*)
chart,proj,offset =isomap(G,dim=1,p=300,verbose = 0) Isomapping of the data return the dim-dimensional ISOMAP chart that best represents the graph G INPUT: - G : Weighted graph that represents the data - dim=1 : number of dimensions - p=300 : nystrom reduction of the problem - verbose = 0: verbosity level OUTPUT

- chart, array of shape(G.V,dim) NOTE: - this 'dev' version is expected to yield more accurate results than the other approximation, because of a better out of samples generalization procedure.

local_correction_for_embedding(*G, chart, sigma=1.0*)
WIP : an unfinished fuction that aims at improving isomap's prbs the idea is to optimize the representation of local distances INPUT: G: the graph to be isomapped chart: the input chart sigma: a scale parameter OUTPUT: chart : the corrected chart

local_sym_normalize(*G*)
graph symmetric normalization; moiover, the normalizing vector is returned NB : this is now in the C graph lib.

mds(*dg, dim=1, verbose=0*)
Multi-dimensional scaling, i.e. derivation of low dimensional representations from distance matrices. INPUT: - dg: a (nbitem,nbitem) distance matrix - dim=1: the dimension of the desired representation - verbose=0: verbosity level

partial_floyd_graph(*G, k*)
Create a graph of the knn in teh geodesic sense, given an input graph G

sparse_local_correction_for_embedding(*G, chart, sigma=1.0, niter=100*)
WIP : an unfinished fuction that aims at improving isomap's prbs the idea is to optimize the representation of local distances INPUT: G: the graph to be isomapped chart: the input chart sigma: a scale parameter OUTPUT: chart : the corrected chart NOTE: the graph G is reordeered

CHAPTER

SIXTYONE

NEUROSPIN.GLM.GLM

61.1 Module: neurospin.glm.glm

Inheritance diagram for nipy.neurospin.glm.glm:

61.2 Classes

61.2.1 contrast

class contrast (*dim, type='t', tiny=1e-50, dofmax=10000000000.0*)

__init__ (*dim, type='t', tiny=1e-50, dofmax=10000000000.0*)
tiny is a numerical constant for computations.

pvalue (*baseline=0.0*)
Return a parametric approximation of the p-value associated with the null hypothesis: (H0) 'contrast equals baseline'

stat (*baseline=0.0*)
Return the decision statistic associated with the test of the null hypothesis: (H0) 'contrast equals baseline'

summary ()
Return a dictionary containing the estimated contrast effect, the associated ReML-based estimation variance, and the estimated degrees of freedom (variance of the variance).

zscore (*baseline=0.0*)
Return a parametric approximation of the z-score associated with the null hypothesis: (H0) 'contrast equals baseline'

61.2.2 glm

class glm(*Y=None, X=None, formula=None, axis=0, model='spherical', method=None, niter=2*)

__init__(*Y=None, X=None, formula=None, axis=0, model='spherical', method=None, niter=2*)

contrast(*c, type='t', tiny=1e-50, dofmax=10000000000.0*)

fit(*Y, X, formula=None, axis=0, model='spherical', method=None, niter=2*)

save(*file*)

61.3 Functions

load(*file*)
Load a fitted glm

ols(*Y, X, axis=0*)
Essentially, compute pinv(X)*Y

CHAPTER

SIXTYTWO

NEUROSPIN.GRAPH.BPMATCH

62.1 Module: neurospin.graph.BPmatch

Routines for Matching of a graph to a cloud of points/tree structures through Bayesian networks (Belief propagation) algorithms

Author: Bertrand Thirion , 2006-2008.

Comment (2009/03/24)

62.2 Functions

BPmatch(*c1, c2, graph, dmax*)

BPmatch_slow(*c1, c2, graph, dmax, imax=20, eps=9.9999999999999998e-13*)
Matching the rows of c1 to those of c2 based on their relative positions graph is a matrix that yields a graph structure on the rows of c1 dmax is measure of the distance decay between points and correspondences for algorithmic details, see Thirion et al, MMBIA 2006

BPmatch_slow_asym(*c1, c2, G1, G2, dmax*)
New version which makes the differences between ascending and descending links - c1 and c2 are arrays of shape (n1,d) and (n2,d) that represent features or coordinates, where n1 and n2 are the number of things to be put in correpondence and d is the common dim - G1 and G2 are corresponding graphs (forests in fff sense) - dmax is a typical distance to compare positions

BPmatch_slow_asym_dep(*c1, c2, G1, G2, dmax*)
New version which makes the differences between ascending and descending links

BPmatch_slow_asym_dev(*c1, c2, G1, G2, dmax*)
New version which makes the differences between ascending and descending links INPUT: - c1 and c2 are arrays of shape (n1,d) and (n2,d) that represent features or coordinates, where n1 and n2 are the number of things to be put in correpondence and d is the common dim - G1 and G2 are corresponding graphs (forests in fff sense) - dmax is a typical distance to compare positions OUTPUT: - (i,j,k): sparse model of the probabilistic relationships, where k is the probability that i is associated with j

EDistance(*X, Y*)
Computation of the euclidian distances between all the columns of X and those of Y

leaves(*G*)

match_trivial(*c1, c2, dmax, eps=9.9999999999999998e-13*)
Matching the rows of c1 to those of c2 based on their relative positions

singles(G)

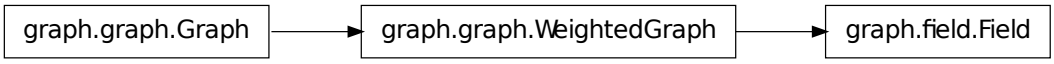
CHAPTER

SIXTYTHREE

NEUROSPIN.GRAPH.FIELD

63.1 Module: neurospin.graph.field

Inheritance diagram for nipy.neurospin.graph.field:



Field processing (smoothing, morphology) routines. Here, a field is defined as arrays [eA,eB,eD,vF] where. (eA,eB,eC) define a graph and vF a dunction defined on the edges of the graph Author: Bertrand Thirion (INRIA Futurs, Orsay, France), 2004-2006.

63.2 Field

class Field(*V, edges=None, weights=None, field=None*)

Bases: nipy.neurospin.graph.graph.WeightedGraph

This is the basic field structure, which contains the weighted graph structure plus a matrix of data (the 'field') -field is an array of size(n,p) where n is the number of vertices of the graph and p is the field dimension

__init__(*V, edges=None, weights=None, field=None*)

INPUT: V: the number of vertices of the graph edges=None: the edge array of the graph weights=None: the asociated weights array field=None: the data itself This is the build function

closing(*nbiter=1*)

self.closing(nbiter=1) Morphological closing of the field IMPUT nbiter=1 : the number of iterations required

constrained_voronoi(*seed*)

label = self.constrained_voronoi(seed) performs a voronoi parcellation of the field starting from the input seed INPUT: seed: int array of shape(p), the input seeds OUTPUT: label: The resulting labelling of the data

copy()

copy function

custom_watershed(*refdim=0, th=-inf*)
idx,depth, major,label = self.custom_watershed(refim = 0,th=-infty) perfoms a watershed analysis of the field. Note that bassins are found aound each maximum (and not minimum as conventionally) INPUT : - th is a threshold so that only values above th are considered by default, th = -infty (numpy) OUTPUT: - idx: the indices of the vertices that are local maxima - depth: the depth of the local maxima depth[idx[i]] = q means that idx[i] is a q-order maximum Note that this is also the diameter of the basins associated with local maxima - major: the label of the maximum which dominates each local maximum i.e. it describes the hierarchy of the local maxima - label : a labelling of thevertices according to their bassin idx, depth and major have length q, where q is the number of bassins label as length n: the number of vertices

diffusion(*nbiter=1*)
diffusion of a field of data in the weighted graph structure INPUT : - nbiter=1: the number of iterations required (the larger the smoother) NB: - This is done for all the dimensions of the field

dilation(*nbiter=1*)
self.dilation(nbiter=1) Morphological openeing of the field IMPUT nbiter=1 : the number of iterations required

erosion(*nbiter=1*)
self.erosion(nbiter=1) Morphological openeing of the field IMPUT nbiter=1 : the number of iterations required

generate_blobs(*refdim = 0, th=-infty, smin=0*)
INPUT - th is a threshold so that only values above th are considered by default, th = -infty (numpy) - smin is the minimum size (in number of nodes) of the blobs to keep.

get_field()

get_local_maxima(*refdim=0, th=-inf*)
idx,depth = get_local_maxima(th=-infty) Look for the local maxima of field[:,refdim] INPUT : - refdim is the field dimension over which the maxima are looked after - th is a threshold so that only values above th are considered by default, th = -infty (numpy) OUTPUT: - idx: the indices of the vertices that are local maxima - depth: the depth of the local maxima : depth[idx[i]] = q means that idx[i] is a q-order maximum

local_maxima(*refdim=0*)
Look for all the local maxima of a field INPUT : - refdim is the field dimension over which the maxima are looked after OUTPUT: - depth: a labelling of the vertices such that

> depth[v] = 0 if v is not a local maximum depth[v] = 1 if v is a first order maximum ... depth[v] = q if v is a q-order maximum

opening(*nbiter=1*)
self.opening(nbiter=1) Morphological openeing of the field IMPUT nbiter=1 : the number of iterations required

print_field()

set_field(*field*)

threshold_bifurcations(*refdim=0, th=-inf*)
idx,height, parents,label = threshold_bifurcations(refdim = 0,th=-infty) perfoms theanalysis of the level sets of the field: Bifurcations are defined as changes in the topology in the level sets when the level (threshold) is varied This can been thought of as a kind of Morse description INPUT : - th is a threshold so that only values above th are considered by default, th = -infty (numpy) OUTPUT: - idx: the indices of the vertices that are local maxima - height: the depth of the local maxima depth[idx[i]] = q means that idx[i] is a q-order maximum Note that this is also the diameter of the basins associated with local maxima - parents: the label of the maximum which dominates each local maximum i.e. it describes the hierarchy of the local maxima - label : a labelling of thevertices according to their bassin idx, depth and major have length q, where q is the number of bassins label as length n: the number of vertices

CHAPTER

SIXTYFOUR

NEUROSPIN.GRAPH.GRAPH

64.1 Module: neurospin.graph.graph

Inheritance diagram for nipy.neurospin.graph.graph:

Graph routines. Author: Bertrand Thirion (INRIA Futurs, Orsay, France), 2004-2008.

64.2 Classes

64.2.1 BipartiteGraph

class BipartiteGraph(*V, W, edges=None, weights=None*)

Bases: nipy.neurospin.graph.graph.WeightedGraph

This is a bipartite graph structure, i.e. a graph there are two types of nodes, such that edges can exist only between nodes of type 1 and type 2 (not within) fields: - V (int,>0) the number of type 1 vertices - W (int,>0) the number of type 2 vertices - E : (int) the number of edges - edges: array of shape (self.E,2) reprensenting pairwise neighbors - weights, array of shape (self.E), +1/-1 for scending/descending links

__init__(*V, W, edges=None, weights=None*)

INPUT: V: the number of edges of the graph edges=None: the edge array of the graph weights=None: the asociated weights array This is the build function

check_feature_matrices(*X, Y*)

self.check_feature_matrices(self,X,Y) checks wether the dismension of X and Y is coherent with self and possibly reshape it INPUT: - The arrays X,Y is assumed to be of size (self.V) or (self.V,p) and (self.W) or (self.W,p) respectively where p = dimension of the features

copy()

G = self.copy() returns a copy of self

cross_eps(*X, Y, eps=1.0*)
E = self.cross_eps(X,Y,eps=1.) set the graph to be the eps-neighbours graph of from X to Y INPUT: - The arrays X,Y is assumed to be of size (self.V) or (self.V,p) and (self.W) or (self.W,p) respectively where p = dimension of the features - eps=1 : the neighbourhood size considered OUTPUT: - the number of edges of the resulting graph, self.E NOTE: - It is assumed that the features are embedded in a (locally) Euclidian space - for the sake of speed it is advisable to give a PCA-preprocessed matrices X and Y.

cross_eps_robust(*X, Y, eps=1.0*)
E = self.cross_eps(X,Y,eps=1.) set the graph to be the eps-neighbours graph of from X to Y this procedure is robust in the sense that for each row of X at least one matching row Y is found, even though the distance is greater than eps. INPUT: - The arrays X,Y is assumed to be of size (self.V) or (self.V,p) and (self.W) or (self.W,p) respectively where p = dimension of the features - eps=1 : the neighbourhood size considered OUTPUT: - the number of edges of the resulting graph, self.E NB: - It is assumed that the features are embedded in a (locally) Euclidian space - for the sake of speed it is advisable to give a PCA-preprocessed matrices X and Y.

cross_knn(*X, Y, k=1*)
E = self.cross_knn(X,Y,k) set the graph to be the k-nearest-neighbours graph of from X to Y INPUT:

•The arrays X,Y is assumed to be of size (self.V) or (self.V,p)

and (self.W) or (self.W,p) respectively where p = dimension of the features - k=1 : is the number of neighbours considered OUTPUT: - the number of edges of the resulting graph, self.E NB: - It is assumed that the features are embedded in a (locally) Euclidian space - for the sake of speed it is advisable to give a PCA-preprocessed matrices X and Y.

set_edges(*edges*)
sets self.edges=edges if 1. edges has a correct size 2. edges take values in [0..V-1]*[0..W-1]

64.2.2 Forest

class Forest(*V, parents=None*)
Bases: nipy.neurospin.graph.graph.WeightedGraph

This is a Forest structure, i.e. a set of trees - the nodes can be segmented into trees - within each tree a node has one parent and children (hierarchical structure) - some of the nodes can be viewed as leaves, other as roots - the edges within a tree are associated with a weight: +1 from child to parent -1 from parent to child

fields: - V : (int,>0) the number of vertices - E : (int) the number of edges - parents: array of shape (self.V) the parent array - edges: array of shape (self.E,2) reprensenting pairwise neighbors - weights, array of shape (self.E), +1/-1 for scending/descending links - children: list of arrays that represents the childs of any node

__init__(*V, parents=None*)
INPUT: V: the number of edges of the graph parents = None: array of shape (V) the parents of the graph by default, the parents are set to range(V), i.e. each node is its own parent, and each node is a tree

all_distances(*seed=None*)
returns all the distanceof the graph as a tree dg = self.all_distances(seed=None) by convention infinte distances are given the distance np.infty

check()
Check that the proposed is indeed a graph, i.e. contains no loop NOTE : slow implementation... to be rewritten in C OUTPUT : a boolean b==0 iff there are loops, 1 otherwise

compute_children()
self.compute_children() define the children list OUPUT: - children: a list of self.V lists that yields the children of each node

define_graph_attributes()
define the edge and weights array

depth_from_leaves()
compute a labelling of the nodes which is 0 for the leaves, 1 for their parents etc and maximal for the roots

get_children(*v=-1*)
returns the list list of children arrays in all the forest if v==-1 or the children of v otherwise

get_descendents(*v*)
returns the nodes that are children of v

isleaf()
returns a bool array of shape(self.V) so that isleaf==1 iff the node is a leaf in the forest (has no kids)

isroot()
returns a bool array of shape(self.V) so that isleaf==1 iff the node is a root in the forest i.e. : is its own parent

leaves_of_a_subtree(*ids, custom=False*)
tests whether the given nodes within ids represent all the leaves of a certain subtree of self if custom==true the behavior of the function is somewhat mroe bizare: - the different connected components are considered as being in a same greater tree - when a node has more than two subbbranches, any subset of these children is considered as a subtree

merge_simple_branches()
merge the branches of the forest that are the only child of the parent branch into their child

reorder_from_leaves_to_roots()
reorder the tree so that the leaves come first then their parents and so on, and the roots are last the permutation necessary to apply to all vertex-based information

subforest(*valid*)
creates a subforest with the vertices for which valid>0 and with the correponding set of edges the children of deleted vertices become their own parent a new forest is created

64.2.3 Graph

class Graph(*V, E=0*)
This is the basic topological (non-weighted) directed Graph class fields : - V(int) = the number of vertices - E(int) = the number of edges - edges = array of int with shape (E,2) : the edges of the graph

__init__(*V, E=0*)

adjacency()

cc()
Returns an array of labels corresponding to the different connex components of the graph. output: -label: array of shape(self.V) that labels the vertices

complete()

degrees()
returns the degree of the graph vertices ouput: - rdegree: array of shape self.V (the right degree) - ldegree: array of shape self.V (the left degree)

get_E()

get_V()

get_edges()

get_vertices()

main_cc()

set_edges(*edges*)
sets self.edges=edges if 1. edges has a correct size 2. edges take values in [1..V]

show(*figid=-1*)
show the graph as a planar graph INPUT: -figid = -1 the figure id in pylab by default a new figure is created OUTPUT: - figid

64.2.4 WeightedGraph

class WeightedGraph(*V, edges=None, weights=None*)
Bases: nipy.neurospin.graph.graph.Graph

This is the basic weighted, directed graph class implemented in fff fields : - V(int) = the number of vertices - E(int) = the number of edges - edges = array of int with shape (E,2) : the edges of the graph - weihghts = array of int with shape (E) : the weights/length of the graph edges

__init__(*V, edges=None, weights=None*)
INPUT: V: the number of edges of the graph edges=None: the edge array of the graph weights=None: the asociated weights array This is the build function

Kruskal()
K = self.Kruskal() creates the MST of self using Kruskal's algo. efficient is self is sparse NOTE: if self contains several connected components, self.Kruskal() will also retain a graph with k connected components

Kruskal_dev()
K = self.Kruskal() creates the MST of self using Kriskal's algo. efficient is self is sparse NOTE: if self contains several connected components, self.Kruskal() will also retain a graph with k connected components

Voronoi_Labelling(*seed*)
label = self.Voronoi_Labelling(seed) performs a voronoi labelling of the graph INPUT: - seed is an array of vertices from which the cells are built OUTPUT: - the labelling of the vertices

Voronoi_diagram(*seeds, samples*)
self.Voronoi_diagram(seeds,samples) Defines the graph as the Voronoi diagram (VD) that links the seeds. The VD is defined using the sample points. INPUT: - seeds: array of shape (self.V,dim) - samples: array of shape (nsamples,dim) NOTE: - by default, the weights are a Gaussian function of the distance - The implementation is somewhat awkward

WeightedDegree(*c*)
returns the sum of weighted degree of graph self INPUT: - c (int) if c==0 considering left side if c==1 considering right side of the edges OUTPUT: - wd : array of shape (self.V): the resulting weighted degree NOTE:inefficient code

adjacency()
INPUT: None OUTPUT: -A : an ((self.V*self.V),np.double) array that represents the adjacency matrix of the graph NOTE: Future version should allow sparse matrix coding

anti_symmeterize()
self.anti_symmeterize() anti-symmeterize the self , ie produces the graph whose adjacency matrix would be the antisymmetric part of its current adjacency matrix

cliques()
cliques = self.cliques() OUTPUT: a labelling of the vertices according to the clique they belong to the cliques are defined using replicator dynamics equations

complete(*) makes self a complete graph (i.e. each pair of vertices is an edge*)

converse_edge()
Returns the index of the edge (j,i) for each edge (i,j) NOTE/TODO :slow and memory-consuming version a C implementation is necessary

copy()
G = self.copy() returns a copy of self

cut_redundancies()
self.cut_redudancies() Remove possibly redundant edges: if an edge (ab) is present twice in the edge matrix, only the first instance in kept. The weights are processed accordingly OUPUT: the number of edges, self.E

dijkstra(*seed=0*)
dg = self.dijkstra(seed=0) returns all the [graph] geodesic distances starting from seed it is mandatory that the graph weights are non-negative INPUT: - seed is the edge from which the distances are computed OUTPUT: - the graph distance dg from the seed to any edge

eps(*X, eps=1.0*)
E = self.eps(X,eps=1.) set the graph to be the eps-nearest-neighbours graph of the data INPUT: - The array X is assumed to be of size (n) or (n,p) where n = number of vertices of the graph and p = dimension of the features - eps=1. : the neighborhood width OUTPUT: - the number of edges of the resulting graph, self.E NB: - It is assumed that the features are embedded in a (locally) Euclidian space - trivial edges (aa) are included - for the sake of speed it is advisable to give a PCA-preprocessed matrix X.

floyd(*seed=None*)
dg = self.floyd(seed=None): returns all the geodesic distances starting from seeds it is mandatory that the graph weights are non-negative INPUT: - seed is an arry of edges from which the distances are computed if seed==None, then every edge is a seed point OUTPUT: - the graph distance dg from each seed to any edge Note that it has size (nbseed,nbedges) NB: - In fact the algo proceeds byr epeating dijkstra's algo for each seed. floyd's algo is not used (O(self.V)^3 complexity...) - by convention, infinte distances are coded with sum(self.wedges)+1

from_3d_grid(*xyz, k=18*)
E = self.from_3d_grid(xyz,k=18) set the graph to be the topological neighbours graph of the dataset xyz, in the k-connectivity scheme INPUT: - xyz: an array of int with shape (n,3), where n=self.V - k = 18: the number of neighbours considered. (6,18 or 26) OUTPUT: - E: the number of edges of the graph

from_adjacency(*A*)
sets the edges of self according to the adjacency matrix M INPUT M: array of size(n,n) so that n = self.V

get_weights()

is_connected()
States whether self is connected or not

knn(*X, k=1*)
E = knn(X,k) set the graph to be the k-nearest-neighbours graph of the data INPUT: - The array X is assumed to be of size (n) or (n,p) where n = number of vertices of the graph and p = dimension of the features - k=1 : is the number of neighbours considered OUTPUT: - the number of edges of the resulting graph, self.E NB: - It is assumed that the features are embedded in a (locally) Euclidian space - the knn system is symmeterized: if (ab) is one of the edges then (ba) is also included - trivial edges (aa) are not included - for the sake of speed it is advisable to give a PCA-preprocessed matrix X.

left_incidence()
return the left incidence matrix of self as a list of lists: i.e. the list[[e.0.0,..,e.0.i(0)],..,[e.V.0,E.V.i(V)]] where e.i.j is the set of edge indexes so that e.i.j[0] = i

list_of_neighbors()
ln = self.list_of_neighbors() returns the set of neighbors as a python list

mst(*X*)
l = self.mst(X) maskes self the MST of the array X INPUT: - X: an array of shape (n,p) where n=self.V and p is the feature dimension OUTPUT: - the length of the mst NB: - It is assumed that the features are embedded in a (locally) Euclidian space - The edge system is symmeterized: if (ab) is one of the edges then (ba) is another edge - As a consequence, the graph comprises (2*self.V-2) edges - the algorithm uses Boruvska's method

normalize(*c=0*)
self.normalize(c=0) Normalize the graph according to the index c Normalization means that the sum of the edges values that go into or out each vertex must sum to 1 INPUT: - c=0 is an index that designates the array according to which D is normalized It is an optional argument, by default c = 0 c == 0 => for each vertex a, sum{edge[e,0]=a} D[e]=1 c == 1 => for each vertex b, sum{edge[e,1]=b} D[e]=1 c == 2 => symmetric ('l2') normalization NB: Note that when sum(edge[e,.]=a) D[e]=0, nothing is performed

remove_edges(*valid) Removes all the edges for which valid==0 INPUT: - valid, an array of shape (self.E*)

remove_trivial_edges()
self.remove_trivial_edges() Removes trivial edges, i.e. edges that are (vv)-like self.weights and self.E are corrected accordingly OUTPUT: The number of edges

reorder(*c=0*)
self.reorder(c=0) Reorder the graph according to the index c INPUT: - c=0 is an index that designates the array according to which the vectors are jointly reordered c == 0 => reordering makes edges[:,0] increasing, and edges[:,1] increasing for edges[:,0] fixed c == 1 => reordering makes edges[:,1] increasing, and edges[:,0] increasing for edges[:,1] fixed c == 2 => reordering makes weights increasing

right_incidence()
return the right incidence matrix of self as a list of lists: i.e. the list[[e.0.0,..,e.0.i(0)],..,[e.V.0,E.V.i(V)]] where e.i.j is the set of edge indexes so that e.i.j[1] = i

set_euclidian(*X) Compute the weights of the graph in an Euclidian space INPUT: X is the coordinate matrix of the vertices it is assumed to be of size (self.V, p*)

set_gaussian(*X, sigma = 0) Compute the weights of the graph as a gaussian function of their length INPUT: - X is the coordinate matrix of the vertices it is assumed to be of size (self.V, p) - sigma = 0 : the parameter of the gaussian function NB = when sigma = 0, an empirical value is used : sigma = sqrt(mean(||X, [self.edges, [:, 0], :], -X, [self.edges, [:, 1], :], ||^2)*)

set_weights(*weights*)
INPUT: -weights : an array of size self.V which yields edges weights

show(*X=None, figid=-1*)
a = self.show(X=None) plots the current graph in 2D INPUT: - X=None, a set of coordinates that can be used to embed the vertices in 2D. to be useful X, should have shape((self.V,p)) if p>2, a svd reduces X for display if p=3 the figure si 3d By default, the graph is presented on a circle - figid=-1: a figure id for pylab plotting by default, a new figure is created OUTPUT: a = figure handle NB: This should be used only for small graphs...

skeleton()
G = self.skeleton() returns a MST that based on self.weights Caveat: self must be connected

subgraph(*valid*)
creates a subgraph with the vertices for which valid>0 and with the correponding set of edges INPUT: - valid of shape (self.V): nonzero for vertices to be retained CAVEAT: - the vertices are renumbered as [1..p] where p = sum(valid>0) - when sum(valid==0) then None is returned

symmeterize()
self.symmeterize() symmeterize the graphself , ie produces the graph whose adjacency matrix would be

the symmetric part of its current adjacency matrix

WeightedGraph.to_neighb() converts the graph to a neighboring system The neighboring s

64.3 Function

concatenate_graphs(*G1, G2*)

G = concatenate(G1,G2) Sets G as the concatenation of the graphs G1 and G2 It is thus assumed that the vertices of G1 and G2 are isjoint sets INPUT: - G1,K2: the two graphs to be concatenated OUPUT - G NOTE: this implies that the vertices of G corresponding to G2 are labeled [G1.V .. G1.V+G2.V]

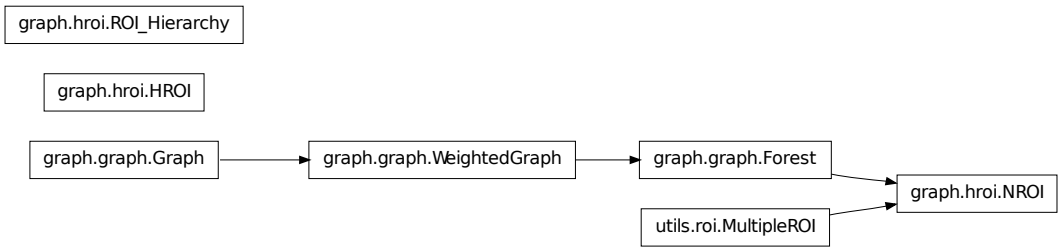
CHAPTER

SIXTYFIVE

NEUROSPIN.GRAPH.HROI

65.1 Module: neurospin.graph.hroi

Inheritance diagram for nipy.neurospin.graph.hroi:



65.2 Classes

65.2.1 HROI

class HROI (*parents=None, header=None, id=None*)

Tentative alternative definition of multiple ROI class

__init__ (*parents=None, header=None, id=None*)

65.2.2 NROI

class NROI (*parents=None, header=None, id=None*)

Bases: nipy.neurospin.utils.roi.MultipleROI, nipy.neurospin.graph.graph.Forest

Class for ntested ROIs. This inherits from both the Forest and MultipleROI

__init__ (*parents=None, header=None, id=None*)

clean (*valid*)

make_forest()
output an fff.forest structure to represent the ROI hierarchy

make_graph()
output an fff.graph stracture to represent the ROI hierarchy

merge_ascending(*valid, methods=None*)
self.merge_ascending(valid) Remove the non-valid items by including them in their parents when it exists methods indicates the way possible features are dealt with

65.2.3 ROI_Hierarchy

class ROI_Hierarchy(*k=1, seed=None, parents=None, label=None*)
Class for the modelling of hierarchical ROIs main attributes: k : number of regions in the graph parents : parents in the tree sense of an ROI seed : reference item(voxel) for a given ROI Label : associated labelling of a certain dataset ROI_features : list of arrays that are ROI-related features ROI_feature_ids : identifiers of the ROI-related features

__init__(*k=1, seed=None, parents=None, label=None*)

argmax(*dmap*)
idx = self.argmax(dmap) INPUT: - dmap: array of shape(np.size(self.label)) OUTPUT: - idx: array of indices with shape (self.k) for each label i in [0..k-1], argmax_{label==i}(map) is computed and the result is returned in idx

check()
check that all the informations are compatible...

clean(*valid*)
remove the non-valid compnents and reorder the ROI list

compute_size()

copy()

get_ROI_feature(*feature_id*)
give the feature associated with a given id, if it ecists

get_k()

get_label()

get_leaf_label()
return self.labels, but with -1 instead of the label when the label does not correpond to a leaf

get_parents()

get_seed()

isfield(*id*)
tests whether a given id is among the current list of ROI_features

isleaf()
Returns a boolean vector of size self.k ==1 iff the item is the parents of nobody

make_feature(*data, id, method='mean'*)
self.make_feature(data,id,method='mean') given a dataset, compute a ROI-based feature through averaging, min or max or cumulative mean INPUT: data = array of shape np.size((self.label)) id = (string) id of the feature method = 'min','mean','max' or 'cumulative_mean'

make_forest()
output an fff.forest structure to represent the ROI hierarchy

make_graph()
output an fff.graph stracture to represent the ROI hierarchy

maxdepth()
depth = self.maxdepth() return a labelling of the nodes so that leaves are labelled by 0 and depth[i] = max_{j in ch[i]} depth[j] + 1 recursively

merge_ascending(*valid, methods=None*)
self.merge_ascending(valid) Remove the non-valid items by including them in their parents when it exists methods indicates the way possible features are dealt with

merge_descending(*methods=None*)
self.merge_descending() Remove the items with only one son by including them in their son methods indicates the way possible features are dealt with

propagate_AND_to_root(*prop) propagates some binary property in the tree that is defined in the leaves so that prop, [parents], = AND(prop, [children]*)

propagate_upward(*label*)
label = self.propagate_upward(label) Assuming that label is a certain positive integer field (i.e. labels) that is defined at the leaves of the tree and can be compared, this propagates these labels to the parents whenever the children nodes have coherent propserties otherwise the parent value is unchanged

reduce_to_leaves()
h2 = reduce_to_leaves(self) create a new set of rois which are only the leaves of self

remove_feature(*feature_id*)
removes the feature associated with a given id, if it exists

set_ROI_feature(*feature, feature_id*)
add a new feature to the class

set_label(*label*)

set_parents(*parents*)

set_seed(*seed*)

subtree(*k*)
l = self.subtree(k) returns an array of the nodes included in the subtree rooted in k

tree_depth()
td = self.tree_depth() return the maximal depth of any node in the tree

65.3 Function

test_nroi(*verbose=0*)

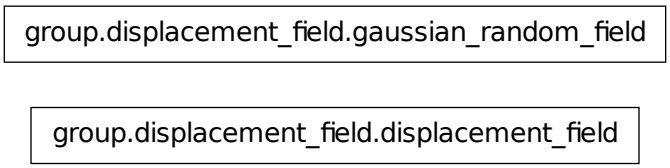
CHAPTER

SIXTYSIX

NEUROSPIN.GROUP.DISPLACEMENT_FIELD

66.1 Module: neurospin.group.displacement_field

Inheritance diagram for nipy.neurospin.group.displacement_field:



66.2 Classes

66.2.1 displacement_field

class displacement_field(*XYZ, sigma, n=1, mask=None, step=None*)

Sampling of multiple vector-valued displacement fields on a 3D-lattice. Displacement fields are generated as linear combinations of fixed displacements. The coefficients are random Gaussian variables.

__init__(*XYZ, sigma, n=1, mask=None, step=None*)

Input : XYZ (3,p) array of voxel coordinates sigma <float> standard deviate of Gaussian filter kernel

Each displacement block has length 4*sigma

n <int> number of generated displacement fields. mask (q,) displacement blocks are limited to mask The constructor creates the following fields : self.block List of N block masks (voxel index vectors) self.weights List of N block weights (same shape as the masks) self.U (3,n,N) Displacement coefficients self.V (3,n,p) Displacements self.W (3,n,p) Discretize displacements self.I (n,p) Displaced voxels index

(voxel k in the mask is displaced by field i to voxel self.I[i,k])

compute_inner_blocks()

Generate self.inner_blocks, index of blocks which are "far from" the borders of the lattice.

init_displacement_blocks()
Called by class constructor

sample(*i, b, proposal='prior', proposal_std=None, proposal_mean=None*)
Generates U, V, L, W, I, where U, V, W, I are proposals for self.U[:,i,b], self.V[:,i,block], self.W[:,i,L], self.I[i,L] if block = self.block[b]. W and I are given only in those voxels, indexed by L, where they differ from current values. Proposal is either 'prior', 'rand_walk' or 'fixed'

sample_all_blocks(*proposal_std=None, proposal_mean=None*)
Generates U, V, W, I, proposals for self.U[:, i], self.V[:, i], self.W[:, i], self.I[i]. Proposal is either 'prior', 'rand_walk' or 'fixed'

66.2.2 gaussian_random_field

class gaussian_random_field(*XYZ, sigma, n=1*)

__init__(*XYZ, sigma, n=1*)

sample(*i, std*)

66.3 Functions

square_gaussian_filter(*input, sigma, output=None, mode='reflect', cval=0.0*)
Multi-dimensional Squared Gaussian filter.

The standard-deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

Note: The multi-dimensional filter is implemented as a sequence of one-dimensional convolution filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

square_gaussian_filter1d(*input, sigma, axis=-1, output=None, mode='reflect', cval=0.0*)
One-dimensional Squared Gaussian filter.

The standard-deviation of the Gaussian filter is given by sigma.

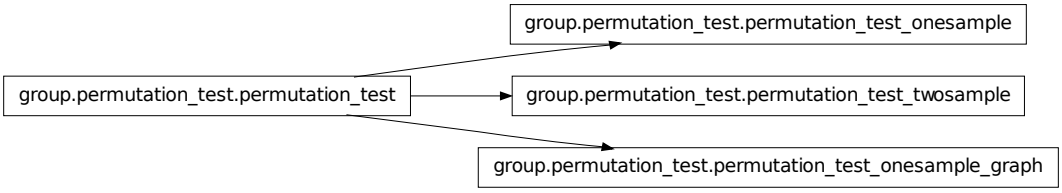
CHAPTER

SIXTYSEVEN

NEUROSPIN.GROUP.PERMUTATION_TEST

67.1 Module: neurospin.group.permutation_test

Inheritance diagram for nipy.neurospin.group.permutation_test:



One and two sample permutation tests.

67.2 Classes

67.2.1 permutation_test

class permutation_test()

This generic permutation test class contains the calibration method which is common to the derived classes permutation_test_onesample and permutation_test_twosample

calibrate(*nperms=10000, clusters=None, cluster_stats=, ['size', 'Fisher'], regions=None, region_stats=, ['Fisher'], verbose=False*)

Calibrate cluster and region summary statistics using permutation test

Parameters **nperms** : int, optional

Number of random permutations generated. Exhaustive permutations are used only if nperms=None, or exceeds total number of possible permutations

clusters : list [(thresh1,diam1),(thresh2,diam2),...], optional

List of cluster extraction pairs: (thresh,diam). *thresh* provides T values threshold, *diam* is the maximum cluster diameter, in voxels. Using *diam*==None yields classical suprathreshold clusters.

cluster_stats : list [stat1,...], optional

List of cluster summary statistics id (either 'size' or 'Fisher')

regions : list [Labels1,Labels2,...]

List of region labels arrays, of size (p,) where p is the number of voxels

region_stats : list [stat1,...], optional

List of cluster summary statistics id (only 'Fisher' supported for now)

verbose : boolean, optional

"Chatterbox" mode switch

Returns **voxel_results** : dict

A dictionary containing the following keys: `p_values` (p,) Uncorrected p-values.''Corr_p_values'' (p,) Corrected p-values, computed by the Tmax procedure. `perm_maxT_values` (nperms) values of the maximum statistic under permutation.

cluster_results : list [results1,results2,...]

List of permutation test results for each cluster extraction pair. These are dictionaries with the following keys "thresh", "diam", "labels", "expected_voxels_per_cluster", "expected_number_of_clusters", and "peak_XYZ" if XYZ field is nonempty and for each summary statistic id "S": "size_values", "size_p_values", "S_Corr_p_values", "perm_size_values", "perm_maxsize_values"

region_results :list [results1,results2,...] :

List of permutation test results for each region labels arrays. These are dictionaries with the following keys: "label_values", "peak_XYZ" (if XYZ field nonempty) and for each summary statistic id "S": "size_values", "size_p_values", "perm_size_values", "perm_maxsize_values"

height_threshold(*pval*)
Return the uniform height threshold matching a given permutation-based P-value.

pvalue(*Tvalues=None*)
Return uncorrected voxel-level pseudo p-values.

zscore(*Tvalues=None*)
Return z score corresponding to the uncorrected voxel-level pseudo p-value.

67.2.2 permutation_test_onesample

class permutation_test_onesample(*data, XYZ, axis=0, vardata=None, stat_id='student', base=0.0, niter=5, ndraws=1000000*)
Bases: nipy.neurospin.group.permutation_test.permutation_test

Class derived from the generic permutation_test class. Inherits the calibrate method

__init__(*data, XYZ, axis=0, vardata=None, stat_id='student', base=0.0, niter=5, ndraws=1000000*)
Initialize permutation_test_onesample instance, compute statistic values in each voxel and under permutation In: data data array

XYZ voxels coordinates axis <int> Subject axis in data

vardata variance (same shape as data) optional (if None, mfx statistics cannot be used)

stat_id <char> choice of test statistic (see onesample.stats for a list of possible stats)

base <float> mean signal under H0 niter <int> number of iterations of EM algorithm ndraws <int> Number of generated random t values

Out: self.Tvalues voxelwise test statistic values self.random_Tvalues sorted statistic values in random voxels and under random

sign permutation

67.2.3 permutation_test_onesample_graph

class permutation_test_onesample_graph(*data, G, axis=0, vardata=None, stat_id='student', base=0.0, niter=5, ndraws=1000000*)

Bases: nipy.neurospin.group.permutation_test.permutation_test

Class derived from the generic permutation_test class. Inherits the calibrate method

__init__(*data, G, axis=0, vardata=None, stat_id='student', base=0.0, niter=5, ndraws=1000000*)

Initialize permutation_test_onesample instance, compute statistic values in each voxel and under permutation In: data data array

G weighted graph (each vertex corresponds to a voxel) axis <int> Subject axis in data vardata variance (same shape as data)

optional (if None, mfx statistics cannot be used)

stat_id <char> choice of test statistic (see onesample.stats for a list of possible stats)

base <float> mean signal under H0 niter <int> number of iterations of EM algorithm ndraws <int> Number of generated random t values

Out: self.Tvalues voxelwise test statistic values self.random_Tvalues sorted statistic values in random voxels and under random

sign permutation

67.2.4 permutation_test_twosample

class permutation_test_twosample(*data1, data2, XYZ, axis=0, vardata1=None, vardata2=None, stat_id='student', niter=5, ndraws=1000000*)

Bases: nipy.neurospin.group.permutation_test.permutation_test

Class derived from the generic permutation_test class. Inherits the calibrate method

__init__(*data1, data2, XYZ, axis=0, vardata1=None, vardata2=None, stat_id='student', niter=5, ndraws=1000000*)

Initialize permutation_test_twosample instance, compute statistic values in each voxel and under permutation In: data1, data2 data arrays

XYZ voxels coordinates axis <int> Subject axis in data

vardata1, vardata2 variance (same shape as data) optional (if None, mfx statistics cannot be used)

stat_id <char> choice of test statistic (see onesample.stats for a list of possible stats)

niter <int> number of iterations of EM algorithm ndraws <int> Number of generated random t values

Out: self.Tvalues voxelwise test statistic values self.random_Tvalues sorted statistic values in random voxels and under random

sign permutation

67.3 Functions

compute_cluster_stats (*Tvalues, labels, random_Tvalues, cluster_stats=, ['size', 'Fisher']*)
size_values, Fisher_values = compute_cluster_stats(Tvalues, labels, random_Tvalues, cluster_stats=["size","Fisher"]) Compute summary statistics in each cluster In: see permutation_test_onesample class docstring Out: size_values Array of size nclust, or None if "size" not in cluster_stats

Fisher_values Array of size nclust, or None if "Fisher" not in cluster_stats

compute_region_stat (*Tvalues, labels, label_values, random_Tvalues*)
Fisher_values = compute_region_stat(Tvalues, labels, label_values, random_Tvalues) Compute summary statistics in each cluster In: see permutation_test_onesample class docstring Out: Fisher_values Array of size nregions

extract_clusters_from_diam (*T, XYZ, th, diam, k=18*)
Extract clusters from a statistical map under diameter constraint and above given threshold In: T (p) statistical map

XYZ (3,p) voxels coordinates th <float> minimum threshold diam <int> maximal diameter (in voxels) k <int> the number of neighbours considered. (6,18 or 26)

Out: labels (p) cluster labels

extract_clusters_from_graph (*T, G, th*)
This returns a label vectorof same size as T, defining connected components for subgraph of weighted graph G containing vertices s.t. T >= th

extract_clusters_from_thresh (*T, XYZ, th, k=18*)
Extract clusters from statistical map above specified threshold In: T (p) statistical map

XYZ (3,p) voxels coordinates th <float> threshold k <int> the number of neighbours considered. (6,18 or 26)

Out: labels (p) cluster labels

max_dist (*XYZ, I, J*)
Maximum distance between two set of points In: XYZ (3,p) voxels coordinates

I (q) index of points J (r) index of points

Out: d <float>

onesample_stat (*Y, V, stat_id, base=0.0, axis=0, Magics=None, niter=5*)
Wrapper for os.stat and os.stat_mfx

peak_XYZ (*XYZ, Tvalues, labels, label_values*)
Returns (3, n_labels) array of maximum T values coordinates for each label value

sorted_values (*a*)
Extract list of distinct sortedvalues from an array

twosample_stat (*Y1, V1, Y2, V2, stat_id, axis=0, Magics=None, niter=5*)
Wrapper for ts.stat and ts.stat_mfx

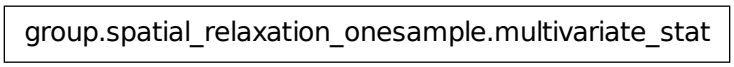
CHAPTER

SIXTYEIGHT

NEUROSPIN.GROUP.SPATIAL_RELAXATION_

68.1 Module: neurospin.group.spatial_relaxation_onesample

Inheritance diagram for nipy.neurospin.group.spatial_relaxation_onesample:



68.2 Class

68.3 multivariate_stat

class multivariate_stat(*data, vardata=None, XYZ=None, std=None, sigma=None, labels=None, network=None, v_shape=0.001, v_scale=0.001, std_shape=0.001, std_scale=0.001, m_mean_rate=0.001, m_var_shape=0.001, m_var_scale=0.001, disp_mask=None, labels_prior=None, label_values=None, labels_prior_mask=None*)

__init__(*data, vardata=None, XYZ=None, std=None, sigma=None, labels=None, network=None, v_shape=0.001, v_scale=0.001, std_shape=0.001, std_scale=0.001, m_mean_rate=0.001, m_var_shape=0.001, m_var_scale=0.001, disp_mask=None, labels_prior=None, label_values=None, labels_prior_mask=None*)

Multivariate modeling of fMRI group data accounting for spatial uncertainty In: data (n,p) estimated effects

vardata (n,p) variances of estimated effects XYZ (3,p) voxel coordinates std <float> Initial guess for standard deviate of spatial displacements sigma <float> regularity of displacement field labels (p,) labels defining regions of interest network (N,) binary region labels (1 for active, 0 for inactive) v_shape <float> intensity variance prior shape v_scale <float> intensity variance prior scale std_shape <float> spatial standard error prior shape std_scale <float> spatial standard error prior scale m_mean_rate <float> mean effect prior rate m_var_shape <float> effect variance prior shape m_var_scale <float> effect variance prior scale disp_mask (q,) mask of the brain, to limit displacements labels_prior (M,r) prior on voxelwise region membership labels_prior_values

(M,r) voxelwise label values where prior is defined labels_prior_mask (r,) Mask of voxels where a label prior is defined

compute_log_likelihood_regionwise(*verbose=False, J=None*)

compute_log_posterior(*v=None, m_mean=None, m_var=None, std=None*)
compute upper bound on log posterior density of model parameters. assuming posterior distribution of hidden variables has been sampled by the 'evaluate' method in 'saem' mode.

compute_log_prior(*v=None, m_mean=None, m_var=None, std=None*)
compute log prior density of model parameters, spatial uncertainty excepted, assuming hidden variables have been initialized

evaluate(*nsimu=1000.0, burnin=100.0, J=None, verbose=False, proposal='prior', proposal_std=None, proposal_mean=None, compute_post_mean=False, mode='saem', update_spatial=True*)
Sample posterior distribution of model parameters, or compute their MAP estimator In: nsimu <int> Number of samples drawn from posterior mean distribution

burnin <int> Number of discarded burn-in samples J (N,) voxel indices where successive mean values are stored verbose <bool> Print some infos during the sampling process proposal <str> 'prior', 'rand_walk' or 'fixed' proposal_mean <float> Used for fixed proposal only proposal_std <float> Used for random walk or fixed proposal mode <str> if mode='saem', compute MAP estimates of model parameters.

if mode='mcmc', sample their posterior distribution

update_spatial <bool> when False, enables sampling conditional on spatial parameters

Out: self.v_values (nsimu+burnin) successive population variance values self.m_values (N, nsimu+burnin) successive mean values (if J is not empty)

if self.labels_prior is not empty: self.labels_post (M,r) posterior distribution of region labels

if self.std is not empty: self.std_values (nsimu+burnin,) successive spatial standard deviate values

if compute_post_mean is True: self.mean_m (p,) posterior average of mean effect self.var_m (p,) posterior variance of mean effect

if self.std is not empty and compute_post_mean is True: self.r (n, nblocks) mean rejection rate for each displacement field self.mean_U (3, n, nblocks) posterior average of displacement weights self.var_U (3, n, nblocks) posterior marginal variances of displacement weights

init_hidden_variables(*mode='saem', init_spatial=True*)

update_block(*i, b, proposal='prior', proposal_std=None, proposal_mean=None, verbose=False*)

update_displacements()

update_effects()

update_labels()

update_mean_effect()

update_parameters_mcmc(*update_spatial=True*)

update_parameters_saem(*update_spatial=True*)

update_summary_statistics(*w, update_spatial=True*)

68.4 Functions

log_gammainv_pdf (*x, a, b*)
log density of the inverse gamma distribution with shape a and scale b, at point x, using Stirling's approximation for a > 100

log_gaussian_pdf (*x, m, v*)
log density of the gaussian distribution with mean m and variance v at point x

CHAPTER

SIXTYNINE

NEUROSPIN.NEURO.AFFINE_REGISTRATION

69.1 Module: neurospin.neuro.affine_registration

69.2 Functions

affine_registration(*source, target, similarity='correlation ratio', interp='partial volume', subsampling=None, normalize=None, search='affine', graduate_search=False, optimizer='powell', resample=True*)

default_subsampling(*dims*)

By default, subsampling factors are tuned so as to match approximately 64**3 voxels.

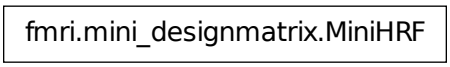
CHAPTER

SEVENTY

NEUROSPIN.NEURO.FMRI.MINI_DESIGNMATR

70.1 Module: neurospin.neuro.fmri.mini_designmatrix

Inheritance diagram for nipy.neurospin.neuro.fmri.mini_designmatrix:



70.2 MiniHRF

class MiniHRF (*peak_hrf=(5.4000000000000004, 10.800000000000001), fwhm_hrf=(5.2000000000000002, 7.3499999999999996), dip=0.34999999999999998, maxderiv=3*)

__init__ (*peak_hrf=(5.4000000000000004, 10.800000000000001), fwhm_hrf=(5.2000000000000002, 7.3499999999999996), dip=0.34999999999999998, maxderiv=3*)

derivParams (*s, paramlist, const=1*)

eval (*x, deriv=0*)

subsample_matrix_example (*timegrid, factorlist, valueslist=None, deriv=False, duration=0.5*)

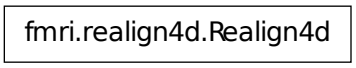
CHAPTER

SEVENTYONE

NEUROSPIN.NEURO.FMRI.REALIGN4D

71.1 Module: neurospin.neuro.fmri.realign4d

Inheritance diagram for nipy.neurospin.neuro.fmri.realign4d:



71.2 Class

71.3 Realign4d

class Realign4d(*img, speedup=4, optimizer='powell'*)

__init__(*img, speedup=4, optimizer='powell'*)

correct_motion()

init_motion_detection(*t*)

The idea is to compute the global variance using the following decomposition:

V = (n-1)/n V1 + (n-1)/n^2 (x1-m1)^2 = alpha + beta d2,

with alpha=(n-1)/n V1, beta = (n-1)/n^2, d2 = (x1-m1)^2.

Only the second term is variable when one image moves while all other images are fixed.

msid(*t*)

resample(*transforms=None*)

resample_all_inmask()

resample_inmask(*t*)

safe_variance(*t*)
No need to invoke self.init_motion_detection.

variance(*t*)

71.4 Functions

grid_coords(*xyz, params, r2v, v2r, transform=None*)

params_to_mat44(*transfo_run, transform=None*)

realign4d(*runs, within_loops=2, bewteen_loops=5, speedup=4, optimizer='powell'*)
Assumes runs is a list of fff2.neuro.fmri_image instance.

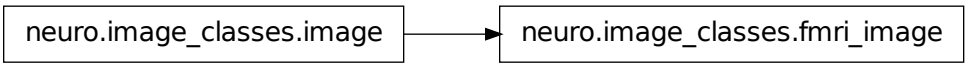
CHAPTER

SEVENTYTWO

NEUROSPIN.NEURO.IMAGE_CLASSES

72.1 Module: neurospin.neuro.image_classes

Inheritance diagram for nipy.neurospin.neuro.image_classes:



Input/output functions. This 'high-level' module is not meant to stay permanently in fff, but is supplied to fff users so that they can process data for real until fff is properly integrated into nipy.

72.2 Classes

72.2.1 fmri_image

class fmri_image(*img, tr=1.0, tr_slices=None, start=0.0, slice_axis=2, slice_order='ascending', interleaved=False*)

Bases: nipy.neurospin.neuro.image_classes.image

__init__(*img, tr=1.0, tr_slices=None, start=0.0, slice_axis=2, slice_order='ascending', interleaved=False*)

inverse_time_transform(*z, tt*)

time_transform(*z, t*)

z_to_slice(*z*)

Account for the fact that slices may be stored in reverse order wrt the scanner coordinate system convention (slice 0 == bottom of the head)

72.2.2 image

class image(*obj=None, transform=None, voxsize=None, iolib='pynifti'*)

__init__ (*obj=None, transform=None, voxsize=None, iolib='pynifti'*)
Basic image class.

transform: 4x4 transformation matrix from array to scanner coordinate system.

IMPORTANT NOTE: the image class assumes that the scanner coordinate system is ALWAYS right-handed and such that x increases from left to right, y increases from posterior to anterior, and z increases in the inferior to superior direction. This transformation is thus exactly what the nifti norm calls the 'qform'.

save (*filename*)
Save an image as described by a dictionary with keys 'array' and voxsize'.

set_array (*array*)

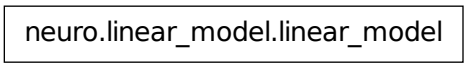
CHAPTER

SEVENTYTHREE

NEUROSPIN.NEURO.LINEAR_MODEL

73.1 Module: neurospin.neuro.linear_model

Inheritance diagram for nipy.neurospin.neuro.linear_model:



73.2 linear_model

class linear_model(*data=None, design_matrix=None, mask=None, formula=None, model='spherical', method=None, niter=2*)

__init__(*data=None, design_matrix=None, mask=None, formula=None, model='spherical', method=None, niter=2*)

contrast(*vector*)
Compute images of contrast and contrast variance.

dump(*filename*)
Dump GLM fit as NPZ file.

affect_inmask(*dest, src, xyz*)

CHAPTER

SEVENTYFOUR

NEUROSPIN.NEURO.SLICES_PLOTER

74.1 Module: neurospin.neuro.slices_ploter

This file contains functions to display a large picture of a bunch of brain slices, using pylab and pynifti. It can be used as a module, or a standalone program. In the later case, see the usage information at the __main__ below. TODO : add an option for non-contour mode and for grid+markers mode

74.2 Functions

display_norm_contour_manyfilenames(*niTemplate, filenameStruct, filenameBold, views=, [0, 1, 2], angles=, [66, 48, 46]*)

Display many brains slices on a large matplotlib figure.

niTemplate : a nifti image defining the "template" size on which other images will be visually resampled, and the contour of which will be overlayed. An in-Talairach mask is ideal here. filenameStruct : a list of structural (or anything else) images filepaths, the view of which is to be drawn on the left part of the final image. They should lie in the same CS as the template. filenameBold : a list of BOLD (or anything else) images filepaths, to be drawn on the right, or None if not needed. They should also lie on the same coordinate system as the template views : a list of views needed (with : 0 or "axial", 1 or "sagittal", 2 or "coronal") angles : a list of slices (respectively on each above-mentionned axes). Caveat : as those are related to the template image, be sure to update it when switching to a different templates.

Note : the implementation is slower than necessary as the same contour is computed many times

resampleNiftiImage(*src, target, order=2, mode='constant'*)

showOneView(*niImage, view, snum, contourNiTemplate=None, nameAsLabels=True, labelcolor='white', contourcolor='yellow', numContour=3, **kargs*)

This matplotlib functions display the slice 'snum' of image 'niImage', for pov 'view' (0 or 'axial', 1 or 'sagittal', 2 or 'coronal'), with a contour overlay (useful to check Registrations). 'contourNiTemplate', if not None, is an image from which the contour overlay is computed, and is assumed to be the same shape as 'niImage', for example, an MNI brainmask. 'nameAsLabels' decorates the axis with english words (such as 'left' or 'anterior') instead of numerical values

CHAPTER

SEVENTYFIVE

NEUROSPIN.NEURO.STATISTICAL_TEST

75.1 Module: neurospin.neuro.statistical_test

75.2 Functions

bonferroni(*p, n*)

cluster_stats(*zimg, mask, height_th, height_control='fpr', cluster_th=0, null_zmax='bonferroni', null_smax=None, null_s=None*)

Return a list of clusters, each cluster being represented by a dictionary. Clusters are sorted by descending size order. Within each cluster, local maxima are sorted by descending depth order.

Input consist of the following: zimg – z-score image mask – mask image height_th – cluster forming threshold height_control – false positive control meaning of cluster forming threshold: 'fpr'|'fdr'|'fwer' size_th – cluster size threshold null_zmax – voxel-level familywise error correction method: 'bonferroni'|'rft'**|array null_smax – cluster-level familywise error correction method: None**|'rft'**|**array null_s – cluster-level calibration method: None|'rft'|array

mask_intersection(*masks*)

Compute mask intersection

onesample_test(*data_images, vardata_images, mask_images, stat_id, comparisons=False, cluster_forming_th=0.01, cluster_th=0*)

prepare_arrays(*data_images, vardata_images, mask_images*)

simulated_pvalue(*t, simu_t*)

z_threshold(*height_th, height_control*)

CHAPTER

SEVENTYSIX

NEUROSPIN.REGISTRATION.REGISTRATION

76.1 Module: neurospin.registration.registration

Inheritance diagram for nipy.neurospin.registration.registration:

76.2 iconic

class iconic(*source, target*)

__init__(*source, target*)

block_voxel_transform(*T*)

eval(*T*)

explore(*ux=, [0], uy=, [0], uz=, [0], rx=, [0], ry=, [0], rz=, [0], sx=, [1], sy=, [1], sz=, [1], qx=, [0], qy=, [0], qz=, [0]*)

optimize(*search='rigid 3D', method='powell', start=None, radius=10*)
radius: a parameter for the 'typical size' in mm of the object being registered. This is used to reformat the parameter vector (translation+rotation+scaling+shearing) so that each element represents a variation in mm.

resample(*T, toresample='source', dtype=None*)

set(*interp='partial volume', similarity='correlation coefficient', normalize=None, subsampling=, [1, 1, 1], corner=, [0, 0, 0], size=None, pdf=None*)

voxel_transform(*T*)

CHAPTER

SEVENTYSEVEN

NEUROSPIN.REGISTRATION.TRANSFORM_AI

77.1 Module: neurospin.registration.transform_affine

77.2 Functions

preconditioner(*radius*)

Computes a scaling vector pc such that, if p=(u,r,s,q) represents affine transformation parameters, where u is a translation, r and q are rotation vectors, and s is a scaling vector, then all components of (p/pc) are somehow comparable and homogeneous to the distance unit implied by the translation component.

transform(*xyz, T*)

T is a 4x4 matrix. xyz is a 3xN array of 3d coordinates stored row-wise.

vector12_to_param(*t, precond, stamp*)

CHAPTER

SEVENTYEIGHT

NEUROSPIN.SCRIPTS.PLOT_ACTIVATION

78.1 Module: `neurospin.scripts.plot_activation`

78.2 Functions

CHAPTER

SEVENTYNINE

NEUROSPIN.SPATIAL_MODELS.BAYESIAN_S

79.1 Module: neurospin.spatial_models.bayesian_structural_analysi

79.2 Functions

CHAPTER

EIGHTY

NEUROSPIN.SPATIAL_MODELS.HIERARCHIC

80.1 Module: `neurospin.spatial_models.hierarchical_parcellation`

80.2 Functions

CHAPTER

EIGHTYONE

NEUROSPIN.SPATIAL_MODELS.PARCEL_IO_I

81.1 Module: `neurospin.spatial_models.parcel_io_nii`

81.2 Functions

CHAPTER

EIGHTYTWO

NEUROSPIN.SPATIAL_MODELS.PARCELLATI

82.1 Module: neurospin.spatial_models.parcellation

Inheritance diagram for nipy.neurospin.spatial_models.parcellation:

Generic Parcellation class: Contains all the items that define a multi-subject parcellation

Author : Bertrand Thirion, 2005-2008

TODO : add a method 'global field', i.e. non-subject-specific info

82.2 Parcellation

class Parcellation (*k, ijk, label, group_labels=None, referential=None, subjects=, []*)

This is the basic Parcellation class: It is defined discretely , i.e. the parcellation is an explicit function on the set of voxels (or equivalently a labelling) we explictly handle the case of multiple subjects, where the labelling varies with the subjects

- k is the number of parcels/classes
- ijk: array of shape(nbvoxels,anatomical_dimension)

that represents the grid of voxels to be parcelled (the same for all subjects) typically anatomical_dimension=3 - referential rerpresents the image referential, resoltuion, position and size this is expressed as an affine (4,4) transformation matrix - label is an (nbvox*subjects) array: nbvox is the number of voxels within the binary mask if the voxel is not labelled in a given subject, then the label is -1 thus the label has integer values in [-1,k-1] - group_labels is a labelling of the template - subjects=none is a list of ids of the subjects by default, is is set as range(self.nb_subj)

__init__ (*k, ijk, label, group_labels=None, referential=None, subjects=, []*)

Constructor

PRFX (*fid, zstat=1, DMtx=None*)

RFX = self.PRFX(fid,zstat=1) Compute the Random effects of the feature on the parcels across subjects

INPUT: - fid is the feature identifier; it is assumed that the feature is 1-dimensional -zstat indicator variable for the output variate if ztsat==0, the basic student statistic is returned if zstat==1, the student stat is converted to a normal(z) variate - DMtx = None : design matrix for the model. So far, it is assumed that DMtx = np.ones(self.nb_subj) OUPUT: - RFX: array with shape (self.k,fdim) containing the parcel-based RFX.

add_subjects (*label, nsubj_id*)
self.add_subjects(label,subj_id) Add some subjects to the structure Not implemented yet.

average_feature (*Feature, subj=-1*)
PF = self.average_feature(Feature) compute parcel-based fetaure bu averaging voxel-based quantities INPUT: - Feature is a list of length self.nb_subj, so that for each s in 0..self.nb_subj-1, that Feature[s] is an (nvox,fdim)-shaped array where nvox is the number of voxels in subject s with label >-1 - subj = -1: subject in which this is performed if subj==-1, this is in all subjects, and it is checked that the fid is not defined yet if subj>-1, this is in one particular subject, and Feature merely is an array, not a list OUPUT: - PF: array of shape (self.nb_subj,self.k,fdim) if subj==-1 or (self.k,fdim) containing the parcel-based features.

boxplot_feature (*pid, fids*)
self.show_feature(pid,fids) This function makes a boxplot of the feature distribution in a given parcel across subjects INPUT: - pid = parcel identifier an integer within the [0..self.K] range - fids = list of features of inetegers

check ()
Some sanity check on the arguments of the class

copy ()
Pa = self.copy() copy method

empty_parcels ()
q = self.empty_parcels() returns the ids of all parcels that are empty

get_feature (*fid*)
self.get_feature(fid): Get feature to the feature list of the structure INPUT: - fid: a string that is the feature id OUTPUT - feature: an array of shape(self.nb_subj,self.k,fdim), where fdim is the feature dimension

isfield (*fid*)
self.isfield(fid) tests whether fid is known as a field

make_feature (*data, fid, subj=-1, method='average'*)
self.make_feature(data,fid,subj=-1,method='average'): Compute and Add a feature to the feature list of the structure INPUT: - data: a list of arrays of shape(nbvoxels,fdim), where fdim is the feature dimension NOTE: if subj>-1, then data is simply an array of shape (nbvoxels,fdim) - fid: a string that is the feature id - subj = -1: subject in which this is performed if subject==-1, this is in all subjects, and it is checked that the fid is not defined yet otherwise, this is in one particular subject, and the feature may be overriden - method = 'average', the way to compute the feature

make_feature_from_info (*fid*)

population ()
pop = self.population() the population of parcellation is the number of voxels included in each parcel this function simply returns an array of shape (number of parcels, number of subjects) that contains the parcel population

remove_feature (*fid*)
self.remove_feature(fid): Remove feature from the feature list of the structure INPUT: - fid: a string that is the feature id

set_feature (*feature, fid*)
self.set_feature(feature,fid): Add a feature to the feature list of the structure INPUT: - feature: an array of shape(self.nb_subj,self.k,fdim), where fdim is the feature dimension - fid: a string that is the feature id

set_group_labels (*glabels*)
self.reset_group_labels(glabels) reset the group labels

set_info (*data, fid*)
self.set_info(data,fid): Add some non-subject specific feature information defined on a voxel-by voxel basis INPUT: - feature: an array of shape(self.nbvox,dim), where dim is the info dimension - fid : an identifier of the information

set_labels (*label) resets the label array of the class INPUT: label = array of shape(self.k, self.nb_subj*)

set_subjects (*subjects*)
self.reset_subjects(subjects) reset the list of subjects name INPUT: - subjects = a list of subjects id with length self.nb_subj

var_feature_intra (*Feature*)
compute the feature variance in each subject and each parcel

variance_inter (*fid*)
HI = self.variance_inter(fid) Compute the variance of the feature at each parcel across subjects INPUT: - fid is the feature identifier OUPUT: - HI

variance_intra (*data, bweight=0*)
Vintra = self.variance_intra(fid) Compute the variance of the feature at each parcel within each subject INPUT: - data is the data on which the variance is estimated: this is a list of arrays - bweight=0: flag for the relative weighting of the parcels if bweight = 1, the variance of each parcels is weighted by its size else, all parcels are equally weighted OUPUT: - VA : array of shape (self.k) of the variance

CHAPTER

EIGHTYTHREE

NEUROSPIN.SPATIAL_MODELS.STRUCTURAL

83.1 Module: neurospin.spatial_models.structural_bfls

Inheritance diagram for nipy.neurospin.spatial_models.structural_bfls:

The main routine of this package that aims at performing the extraction of ROIs from multisubject dataset using the localization. This has been puclished in Thirion et al. Structural Analysis of fMRI Data Revisited: Improving the Sensitivity and Reliability of fMRI Group Studies. IEEE TMI 2007

Author : Bertrand Thirion, 2006-2008

83.2 Class

83.3 Amers

class Amers (*k, subj=None, idx=None, coord=None*)

This class is intended to represent inter-subject regions its members are: - k: (int) the number of regions involved - subj: array of int of size k, an identifier from where ('the subjects') the regions come from - idx : array of int of size k, an index related to each region, typically a seed voxel of the region in a certain representation - coord: array of double of size k*p, where p is a certain space dimension representing coordinates of the regions in a certain unspecified, but common system

__init__ (*k, subj=None, idx=None, coord=None*)

center ()

c = self.center() returns the average of the coordinates

confidence_region (*cs, dmax=10, pval=0.94999999999999996*)

i = self.confidence_region(cs,dmax = 10,pval = 0.95) Sample the pval-confidence region of AF on the proposed coordiante set cs, assuming a Gaussian shape INPUT: - cs: an array of size(n*p) a set of input

coordinates - dmax=10 : an upper bound for the spatial variance to avoid degenerate variance - pval=0.95 cutoff for the CR OUPUT: i = set of entries of the coordinates that are within the cr

homogeneity()
d = self.homogeneity() OUTPUT: d (float) returns the mean distance between the coordinates of the regions

83.4 Functions

Build_Amers(*BF, u, ths=0*)
Given a list of hierarchical ROIs, and an associated labelling, this creates an Amer structure wuch groups ROIs with the same label. INPUT: - BF is the list of hierarchical ROIs. it is assumd that each list corresponds to one subject - u is a labelling array of size the total number of ROIs - ths=0 defines the condition (c): A label should be present in ths subjects in order to be valid OUTPUT: - AF : a list of Amers, each of which describing a cross-subject set of ROIs - newlabel : a relabelling of the individual ROIs, similar to u, which discards labels that do not fulfill the condition (c)

Compute_Amers(*Fbeta, Beta, tal, dmax=10.0, thr=3.0, ths=0, pval=0.20000000000000001*)
This is the main function for contrsucting the BFLs INPUT - Fbeta : field structure that contains the spatial nodes of the dataset - Beta: functional data matrix of size (nbnodes,nbsubj) - tal: spatial coordinates of the nodes (e.g. MNI coords) - dmax=10.: spatial relaxation allowed in the preocedure - thr = 3.0: thrshold at the first-level - ths = 0, number of subjects to validate a BFL - pval = 0.2 : significance p-value for the spatial inference OUPUT - crmap: the map of CR of the activated clusters in the common space - AF : group level BFLs - BFLs: list of first-level nested ROIs - Newlabel: labelling of the individual ROIs

RD_cliques(*Gc, bstochastic=1*)
Replicator dynamics graph segmentation: python implementation INPUT : - Gc graph to be segmented - bstochastic=1 stochastic initialization of the graph OUPUT: - labels : array of size V, the number of vertices of Gc the labelling of the vertices that represent the segmentation

clean_density(*BFLs, dmax, xyz, pval=0.050000000000000003, verbose=0, dev=0, nrec=5, nsamples=10*)
Computation of the positions where there is a significant spatial accumulation of pointwise ROIs. The significance is taken with respect to a uniform distribution of the ROIs. INPUT: - BFLs : a list of ROIs hierarchies, putatively describing ROIs from different subjects CAVEAT 1: The ROI_Hierarchy must have a 'position' feature defined beforehand CAVEAT 2: the structure is edited and modified by this function - dmax : the kernel width (std) for the spatial density estimator - xyz : a set of coordinates on which the test is perfomed it should be written as (nbitems,dimension) array - pval=0.05: corrected p-value for the significance of the test Importantly, the p-value is corrected only for the number of ROIs per subject - verbose=0: verbosity mode - dev=0. If dev=1, a different technique is used for density estimation (WIP) - nrec=5: number of recursions in the test: When some regions fail to be significant at one step, the density is recomputed, and the test is performed again and so on

clean_density_redraw(*BFLs, dmax, xyz, pval=0.050000000000000003, verbose=0, dev=0, nrec=5, nsamples=10*)
Computation of the positions where there is a significant spatial accumulation of pointwise ROIs. The significance is taken with respect to a uniform distribution of the ROIs. INPUT: - BFLs : a list of ROIs hierarchies, putatively describing ROIs from different subjects CAVEAT 1: The ROI_Hierarchy must have a 'position' feature defined beforehand CAVEAT 2: the structure is edited and modified by this function - dmax : the kernel width (std) for the spatial density estimator - xyz : a set of coordinates on which the test is perfomed it should be written as (nbitems,dimension) array - pval=0.05: corrected p-value for the significance of the test Importantly, the p-value is corrected only for the number of ROIs per subject - verbose=0: verbosity mode - dev=0. If dev=1, a different technique is used for density estimation (WIP) - nrec=5: number of recursions in the test: When some regions fail to be significant at one step, the density is recomputed, and the test is performed again and so on 19/02/07: new version without the monotonic exclusion heuristic

compute_density(*BFLs, xyz, dmax*)
Computation of the density of the BFLs points in the xyz volume dmax is a scale parameter

compute_density_dev(*BFLs, xyz, dmax*)
Computation of the density of the BFLs points in the xyz volume dmax is a scale parameter

compute_surrogate_density(*BFLs, xyz, dmax, nsamples=1*)
Cross-validated estimation of random samples of the uniform distributions INPUT: - BFLs : a list of sets of ROIs the list length, nsubj, is taken as the number of subjects - xyz (gs,3) array: a sampling grid to estimate spatial distribution - dmax kernel width of the density estimator - nsamples=1: number of surrogate smaples returned OUTPUT: - surweight: a (gs*nsamples,nsubj) array of samples

compute_surrogate_density_dev(*BFLs, xyz, dmax, nsamples=1*)
caveat: does not work for nsamples>1 This function computes a surrogate density using graph diffusion techniques

fig_density(*sweight, surweight, pval, nlm*)
Plot the histogram of sweight across the image and the thresholds implied by the surrogate model (surweight)

hierarchical_asso(*BF, dmax*)
Compting an association graph of the ROIs defined across different subjects INPUT: - BF a list of ROI hierarchies, one for each subject - dmax : spatial scale used xhen building associtations OUPUT: - G a graph that represent probabilistic associations between all cross-subject pairs of regions. Note that the probabilities are normalized on a within-subject basis.

merge(*Gc, labels*)
Given a first labelling of the graph Gc, this function builds a reduced graph by merging the vertices according to the labelling INPUT: - Gc the input graph - labels : array of size V, the number of vertices of Gc the labelling of the vertices that represent the segmentation OUTPUT: - labels : the new labelling after further merging - Gr the reduced graph after merging

segment_graph_rd(*Gc, nit=1, verbose=0*)
This function prefoms a hard segmentation of the graph Gc using a replicator dynamics approach. The clusters obtained in the first pass are further merged during a second pass, based on a reduced graph INPUT: - Gc : the graph to be segmented OUTPUT: - u : array of size V, the number of vertices of Gc the labelling of the vertices that represent the segmentation

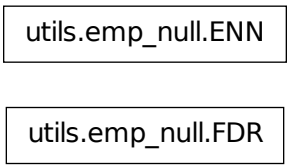
CHAPTER

EIGHTYFOUR

NEUROSPIN.UTILS.EMP_NULL

84.1 Module: neurospin.utils.emp_null

Inheritance diagram for nipy.neurospin.utils.emp_null:



this module contains a class that fits a gaussian model to the central part of an histogram, following schwartzman et al, 2009. This is typically necessary to estimate a fdr when one is not certain that the data behaves as a standard normal under H_0.

Author : Bertrand Thirion, 2008-2009

84.2 Classes

84.2.1 ENN

class ENN (*x*)

Class to compute the empirical null normal fit to the data.

The data which is used to estimate the FDR, assuming a gaussian null from Schwartzmann et al., NeuroImage 44 (2009) 71–82

__init__ (*x*)

Initiate an empirical null normal object.

Parameters **x** : 1D ndarray

The data used to estimate the empirical null.

fdr (*theta*)

given a threshold theta, find the estimated fdr

fdrcurve()
Returns the fdr associated with any point of self.x

learn(*left=0.20000000000000001, right=0.80000000000000004*)
Estimate the proportion, mean and variance of a gaussian distribution for a fraction of the data

Parameters **left** : float, optional

Left cut parameter to prevent fitting non-gaussian data

right : float, optional

Right cut parameter to prevent fitting non-gaussian data

Notes

This method stores the following attributes:

- mu = mu
- p0 = min(1, np.exp(lp0))
- sqsigma : standard deviation of the estimated normal distribution
- sigma = np.sqrt(sqsigma) : variance of the estimated normal distribution

plot(*efp=None, alpha=0.050000000000000003, bar=1*)
plot the histogram of x

Parameters **efp** : float, optional

The empirical fdr (corresponding to x) if efp==None, the false positive rate threshod plot is not drawn.

alpha : float, optional

The chosen fdr threshold

threshold(*alpha=0.050000000000000003, verbose=0*)
Compute the threshold correponding to an alpha-level fdr for x

Parameters **alpha** : float, optional

the chosen false discovery rate threshold.

verbose : boolean, optional

the verbosity level, if True a plot is generated.

uncorrected_threshold(*alpha=0.001, verbose=0*)
Compute the threshold correponding to a specificity alpha for x

Parameters **alpha** : float, optional

the chosen false discovery rate threshold.

verbose : boolean, optional

the verbosity level, if True a plot is generated.

Results :

theta: float :

the critical value associated with the provided p-value

84.2.2 FDR

class FDR(*x*)

This is the basic class to handle false discovery rate computation parameter: fdr.x the samples from which the fdr is derived x is assumed to be a normal variate

The Benjamini-Horchberg procedure is used

__init__(*x*)

x is assumed to be a 1-d array

all_fdr(*x=None, verbose=0*)

Returns all the FDR (false discovery rates) values for the sample x

Parameters **x** : ndarray of shape (n)

The normal variates

all_fdr_from_pvals(*pv, verbose=0*)

Returns the fdr associated with each the values

Parameters **pv** : ndarray of shape (n)

The samples p-value

Returns **q** : array of shape(n)

The corresponding fdrs

check_pv(*pv*)

Do some basic checks on the pv array: each value should be within [0,1]

Parameters **pv** : array of shape (n)

The sample p-values

Returns **pv** : array of shape (n)

The sample p-values

pth_from_pvals(*pv, alpha=0.050000000000000003*)

Given a set pv of p-values, returns the critical p-value associated with an FDR alpha

Parameters **alpha** : float

The desired FDR significance

pv : array of shape (n)

The samples p-value

Returns **pth: float** :

The p value corresponding to the FDR alpha

threshold(*alpha=0.050000000000000003, x=None*)

Given an array x of normal variates, this function returns the critical p-value associated with alpha. x is explicitly assumed to be normal distributed under H_0

Parameters **alpha: float, optional** :

The desired significance, by default 0.05

x : ndarray, optional

The variate. By default self.x is used

Returns **th** : float

The threshold in variate value

threshold_from_student(*df, alpha=0.050000000000000003, x=None*)

Given an array t of student variates with df dofs, returns the critical p-value associated with alpha.

Parameters **df** : float

The number of degrees of freedom

alpha : float, optional

The desired significance

x : ndarray, optional

The variate. By default self.x is used

Returns **th** : float

The threshold in variate value

CHAPTER

EIGHTYFIVE

NEUROSPIN.UTILS.MASK

85.1 Module: `neurospin.utils.mask`

85.2 Functions

computeMaskIntra(*inputFilename, outputFilename, copyFilename=None, m=0.20000000000000001, M=0.90000000000000002, cc=1*)

Depreciated, see compute_mask_intra.

computeMaskIntraArray(*volumeMean, firstVolume, m=0.20000000000000001, M=0.90000000000000002, cc=1*)

Depreciated, see compute_mask_intra.

compute_mask(*mean_volume, reference_volume=None, m=0.20000000000000001, M=0.90000000000000002, cc=1*)

Compute a mask file from fMRI data in 3D or 4D ndarrays.

Compute and write the mask of an image based on the grey level This is based on an heuristic proposed by T.Nichols: find the least dense point of the histogram, between fractions m and M of the total image histogram.

In case of failure, it is usually advisable to increase m.

Parameters **mean_volume** : 3D ndarray

mean EPI image, used to compute the threshold for the mask.

reference_volume: 3D ndarray, optional :

reference volume used to compute the mask. If none is give, the mean volume is used.

m : float, optional

lower fraction of the histogram to be discarded.

M: float, optional :

upper fraction of the histogram to be discarded.

cc: boolean, optional :

if cc is True, only the largest connect component is kept.

Returns **mask** : 3D boolean ndarray

The brain mask

compute_mask_files(*input_filename, output_filename=None, return_mean=False, copy_filename=None, m=0.20000000000000001, M=0.90000000000000002, cc=1*)

Compute a mask file from fMRI nifti file(s)

Compute and write the mask of an image based on the grey level This is based on an heuristic proposed by T.Nichols: find the least dense point of the histogram, between fractions m and M of the total image histogram.

In case of failure, it is usually advisable to increase m.

Parameters **input_filename** : string

nifti filename (4D) or list of filenames (3D).

output_filename : string or None, optional

path to save the output nifti image (if not None).

return_mean : boolean, optional

if True, and output_filename is None, return the mean image also, as a 3D array (2nd return argument).

copy_filename : string, optional

optionally, a copy of the original data saved as a single-file 4D nifti volume.

m : float, optional

lower fraction of the histogram to be discarded.

M: float, optional :

upper fraction of the histogram to be discarded.

cc: boolean, optional :

if cc is True, only the largest connect component is kept.

Returns **mask** : nifti.NiftiImage object

The brain mask

mean_image : 3d ndarray, optional

The main of all the images used to estimate the mask. Only provided if *return_mean* is True.

compute_mask_intra(*input_filename, output_filename=None, return_mean=False, copy_filename=None, m=0.20000000000000001, M=0.90000000000000002, cc=1*)

See compute_mask_files.

compute_mask_intra_array(*volume_mean, reference_volume=None, m=0.20000000000000001, M=0.90000000000000002, cc=True*)

Depreciated, see compute_mask.

compute_mask_sessions(*session_files, m=0.20000000000000001, M=0.90000000000000002, cc=1, threshold=0.5*)

Compute a common mask for several sessions of fMRI data.

Uses the mask-finding algorithmes to extract masks for each session, and then keep only the main connected component of the a given fraction of the intersection of all the masks.

Parameters **session_files** : list of list of strings

A list of list of nifti filenames. Each inner list represents a session.

threshold : float, optional

the inter-session threshold: the fraction of the total number of session in for which a voxel must be in the mask to be kept in the common mask. threshold=1 corresponds to keeping the intersection of all masks, whereas threshold=0 is the union of all masks.

m : float, optional

lower fraction of the histogram to be discarded.

M: float, optional :

upper fraction of the histogram to be discarded.

cc: boolean, optional :

if cc is True, only the largest connect component is kept.

Returns **mask** : 3D boolean ndarray

The brain mask

CHAPTER

EIGHTYSIX

NEUROSPIN.UTILS.NOSETESTER

86.1 Module: neurospin.utils.nosetester

Inheritance diagram for nipy.neurospin.utils.nosetester:

Nose test running

Implements test and bench functions for modules.

86.2 Class

86.3 NoseTester

class NoseTester (*package=None*)

Bases: object

Nose test runner.

Usage: NoseTester(<package>).test()

<package> is package path or module Default for package is None. A value of None finds calling module path.

Typical call is from module __init__, and corresponds to this:

test = NoseTester().test

This class is made available as numpy.testing.Tester:

from scipy.testing import Tester test = Tester().test

__init__ (*package=None*)

Test class init

Parameters **package** : string or module

If string, gives full path to package If None, extract calling module path Default is None

bench (*label='fast', verbose=1, extra_argv=None*)
Run benchmarks for module using nose

Parameters **label** : {'fast', 'full', '', attribute identifer}

Identifies benchmark to run. This can be a string to pass to the nosetests executable with the'-A' option, or one of several special values. Special values are: 'fast' - the default - which corresponds to

nosetests -A option of 'not slow'.

'full' - fast (as above) and slow benchmark as in no -A option to nosetests - same as ''

None or '' - run all benchmarks attribute_identifier - string passed directly to

nosetests as '-A'

verbose : integer

verbosity value for test outputs, 1-10

extra_argv : list

List with any extra args to pass to nosetests

test (*label='fast', verbose=1, extra_argv=None, doctests=False, coverage=False, **kwargs*)
Run tests for module using nose

Parameters **label** : {'fast', 'full', '', attribute identifer}

Identifies test to run. This can be a string to pass to the nosetests executable with the'-A' option, or one of several special values. Special values are: 'fast' - the default - which corresponds to

nosetests -A option of 'not slow'.

'full' - fast (as above) and slow test as in no -A option to nosetests - same as ''

None or '' - run all tests attribute_identifier - string passed directly to

nosetests as '-A'

verbose : integer

verbosity value for test outputs, 1-10

extra_argv : list

List with any extra args to pass to nosetests

doctests : boolean

If True, run doctests in module, default False

coverage : boolean

If True, report coverage of NumPy code, default False (Requires the coverage module:

http://nedbatchelder.com/code/modules/coverage.html)

86.4 Functions

import_nose()
Import nose only when needed.

run_module_suite(*file_to_run=None*)

skipif(*skip_condition, msg=None*)
Make function raise SkipTest exception if skip_condition is true

Parameters **skip_condition** : bool

Flag to determine whether to skip test (True) or not (False)

msg [string] Message to give on raising a SkipTest exception

Returns **decorator** : function

Decorator, which, when applied to a function, causes SkipTest to be raised when the skip_condition was True, and the function to be called normally otherwise.

Notes

You will see from the code that we had to further decorate the decorator with the nose.tools.make_decorator function in order to transmit function name, and various other metadata.

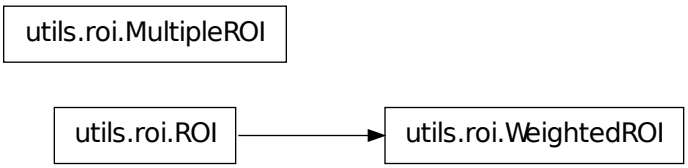
CHAPTER

EIGHTYSEVEN

NEUROSPIN.UTILS.ROI

87.1 Module: neurospin.utils.roi

Inheritance diagram for nipy.neurospin.utils.roi:



87.2 Classes

87.2.1 MultipleROI

class **MultipleROI**(*id='roi', k=0, header=None*)

This is a class to deal with multiple ROIs defined in a given space mroi.header is assumed to provide all the referential information (this should be changed in the future), so that the mroi is basically defined as a multiple sets of 3D coordinates finally, there is an associated feature dictionary. Typically it is assumed that each feature is an (roi,feature_dim) array, i.e. each roi is assumed homogeneous wrt the feature In the future, it might be possible to complexify the structure to model within-ROI variance

__init__(*id='roi', k=0, header=None*)

roi = MultipleROI(id='roi', header=None) - id (string): roi identifier - k: number of rois that are included in the structure - header (nipy header) : referential-defining information

append_balls(*position, radius*)

idem self.as_multiple_balls, but the ROIs are added

as_multiple_balls(*position, radius*)

self.as_multiple_balls(position, radius) Given a set of positions and radii, defines one roi at each (position/radius) couple INPUT: position: array of shape (k,3): the set of positions radius: array of shape (k): the set of radii

check_features()
check that self.features have the coorect size i.e; f.shape[0]=self.k for f in self.features

check_header(*image*)
checks that the image is in the header of self INPUT: - image: (string) the path of an image

clean(*valid*)
remove the regions for which valid<=0

complete_feature(*fid, values*)
completes a feature by appending the values

from_labelled_image(*image, labels=None, add=True*)
All the voxels of the image that have non-zero-value self.k becomes the number of values of the (discrete) image INPUT: - image (string): a nifti label (discrete valued) image -labels=None : the set of image labels that shall be used as ROI definitions By default, all the image labels are used note that this can be used to append features, when rois are already defined

get_size()
return the number of voxels per ROI in one array

make_image(*name*)
write a int nifty image where the nonzero values are the ROIs INPUT: - the desired image name NOTE: - the background values are set to -1 - the ROIs values are set as [0..self.k-1]

plot_feature(*fid*)
boxplot the feature within the ROI Note that this assumes a 1-d feature

set_feature_from_image(*fid, image, method='average'*)
extract some roi-related information from an image INPUT: - fid: feature id - image(string): image name - method='average' (string) : take the roi feature as the average feature over the ROI

set_roi_feature(*fid, data*)
INPUT: - fid (string): feature identifier, e.g. - data: array of shape(self.k,p),with p>0 this function simply stores data

87.2.2 ROI

class ROI(*id='roi', header=None*)
Temporary ROI class for fff Ultimately, it should be merged with the nipy class

ROI definition requires - an identifier - an header (exactly a nifti header at the moment, though not everything is necessary) The ROI can be derived from a image or defined in the coordinate system implied by header.sform()

roi.features is a dictionary of informations on the ROI elements. It is assumed that the ROI is sampled on a discrete grid, so that each feature is in fact a (voxel,feature_dimension) array

__init__(*id='roi', header=None*)
roi = ROI(id='roi', header=None) - id (string): roi identifier - header (nipy header) : referential-defining information

check_header(*image*)
checks that the image is in the header of self INPUT: - image: (string) the path of an image

from_binary_image(*image*)
Take all the <>0 sites of the image as the ROI INPUT: - image: (string) the path of an image

from_labelled_image(*image, label*)
All the voxels of the image that have the pre-defined label INPUT: image: a nifti label (discrete valued) image label (int): the desired label

from_position (*position, radius*)
a ball in the grid requires that the grid and header are defined

from_position_and_image (*image, position*)
the label on the image that is closest to the provided position INPUT: - image: a nifti label (discrete valued) image - position: a position in the common space NOTE: everything could be performed in the image space

get_feature (*fid*)
return the feature corrsponding to fid, if it exists

make_image (*name*)
write a binary nifty image where the nonzero values are the ROI mask INPUT: the desired image name

plot_feature (*fid*)
boxplot the feature within the ROI

representative_feature (*fid, method='mean'*)
Compute a statistical representative of the within-ROI feature

set_feature (*fid, data*)
INPUT: - fid (string): feature identifier, e.g. - data (array of shape (self.VolumeExtent)) this function creates a reduced feature array corresponding to the ROI item OUTPUT: - ldata: array of shape (roi.nbvox,dim) the ROI-based feature

set_feature_from_image (*fid, image*)
extract some roi-related information from an image INPUT: - fid: feature id - image(string): image name

set_feature_from_masked_data (*fid, data, mask*)
idem set_feature but the input data is thought to be masked

87.2.3 WeightedROI

class WeightedROI (*id='roi', header=None, grid=None*)
Bases: nipy.neurospin.utils.roi.ROI

ROI where a weighting is defined on the voxels

__init__ (*id='roi', header=None, grid=None*)

87.3 Functions

test1 (*verbose=0*)

test2 (*verbose=0*)

test_mroi1 (*verbose=0*)

test_mroi2 (*verbose=0*)

test_mroi3 (*verbose=0*)

CHAPTER

EIGHTYEIGHT

NEUROSPIN.UTILS.SIMUL_2D_MULTISUBJEC

88.1 Module: neurospin.utils.simul_2d_multisubject_fmri_dataset

This module conatins a function to produce a dataset which simulates a collection of 2D images This dataset is saved as a 3D nifti image (each slice being a subject) and a 3D array

example of use: make_surrogate_array(nbsubj=1,fid="/tmp/toto.dat",verbose=1)

todo: rewrite it as a class

Author : Bertrand Thirion, 2008-2009

88.2 Functions

cone(*shape, ij, pos, ampli, width*)

Define a cone of the proposed grid

make_surrogate_array(nbsubj=10, dimx=30, dimy=30, sk=1.0, noise_level=1.0, pos=array([[6, [10, 10], ()
[15, 10]]), ampli=array([3, 4, 4]), spatial_jitter=1.0, signal_jitter=1.0, width=5.0, out_t

Create surrogate (simulated) 2D activation data with spatial noise.

Parameters nbsubj: integer, optionnal :

The number of subjects, ie the number of different maps generated.

dimx: integer, optionnal :

The x size of the array returned.

dimy: integer :

The y size of the array returned.

sk: float, optionnal :

Amount of spatial noise smoothness.

noise_level: float, optionnal :

Amplitude of the spatial noise. amplitude=noise_level)

pos: 2D ndarray of integers, optionnal :

x, y positions of the various simulated activations.

ampli: 1D ndarray of floats, optionnal :

Respective amplitude of each activation

spatial_jitter: float, optionnal :

Random spatial jitter added to the position of each activation, in pixel.

signal_jitter: float, optionnal :

Random amplitude fluctuation for each activation, added to the amplitude specified by ampli

width: float or ndarray, optionnal :

Width of the activations

out_text_file: string or None, optionnal :

If not None, the resulting array is saved as a text file with the given file name

out_niftifile: string or None, optionnal :

If not None, the resulting is saved as a nifti file with the given file name.

verbose: boolean, optionnal :

If verbose is true, the data for the last subject is plotted as a 2D image.

Returns dataset: 3D ndarray :

The surrogate activation map, with dimensions (nbsubj, dimx, dimy)

CHAPTER

EIGHTYNINE

NEUROSPIN.UTILS.SMOOTHING

89.1 Module: neurospin.utils.smoothing

Routine for smoothing data using diffusion on graphs. Works well only for small kernels.

Mybe should be abandoned

Author : Bertrand Thirion, 2006-2009

89.2 Functions

cartesian_smoothing(*ijk, data, sigma*)

Smoothing data on a(n uncomplete) cartesian grid INPUT: -ijk : list of the positions, which is assumed to be an (n,3) int array it is typically returned by transpose(numpy.where()) - data : an array of data sampled from the grid size(ijk.shape[0],d) where d is the datas dimension -sigma : the kernel parameter OUTPUT - data, which is the smoothed data

check_smoothing()

CHAPTER

NINETY

NEUROSPIN.UTILS.THRESHOLD

90.1 Module: neurospin.utils.threshold

This module contains the function that thresholds an image and retains only the clusters of size>smin

Author : Bertrand Thirion, 2009

90.2 Functions

threshold_scalar_image(*iimage, oimage, th=0.0, smin=0, mask_image=None*)

this function takes a 'grey level' threshold and a size threshold and gives as output an image where only the suprathreshold component of size > smin have not been thresholded out INPUT: - iimage : the path of a scalar nifti image - oimage: the path of the dcalar output nifti image - th = 0. the chose trheshold -smin=0 the cluster size threshold -mask_image=None: a mask image to determine where in image this applies if mask_image==None, the function is implied on where(image) OUTPUT: - oimage: the output image

threshold_z_image(*iimage, oimage, corr=None, pval=None, smin=0, mask_image=None, method=None*)

this function takes a presumably gaussian image threshold and a size threshold and gives as output an image where only the suprathreshold component of size > smin have not been thresholded out This corresponds to a one-sided classical test the null hypothesis can be take to be the standard normal or the empiricall null. INPUT: - iimage : the path of a presumably z-variate input nifti image - oimage: the path of the output image - corr=None: the correction for multiple comparison method corr can be either None or 'bon' (Bonferroni) or 'fdr' - pval=none: the disired classical p-value. the default behaviour of pval depends on corr if corr==None then pval = 0.001 else pval = 0.05 - smin=0 the cluster size threshold - mask_image=None: a mask image to determine where in image this applies if mask_image==None, the function is implied on where(image) - method=None: model of the null distribution: if method==None: standard null if method=='emp': empirical null OUTPUT: - oimage: the output image

CHAPTER

NINETYONE

NEUROSPIN.UTILS.ZSCORE

91.1 Module: neurospin.utils.zscore

zscore(*pvalue*)
Return the corresponding Z score for a given pvalue.

CHAPTER

NINETYTWO

TESTING.DECORATORS

92.1 Module: testing.decorators

Use numpy testing framework which is based on nose as of v1.2

Extend the decorators to use nipy's gui and data labels.

92.2 Functions

knownfailure(*f*)

make_label_dec(*label, ds=None*)

Factory function to create a decorator that applies one or more labels.

Parameters label : string or sequence One or more labels that will be applied by the decorator to the functions

it decorates. Labels are attributes of the decorated function with their value set to True.

Keywords ds : string An optional docstring for the resulting decorator. If not given, a default docstring is auto-generated.

Returns A decorator.

Examples

```
>>> from nipy.testing import make_label_dec
>>> slow = make_label_dec('slow')
>>> print slow.__doc__
Labels a test as 'slow'


>>> from nipy.testing import make_label_dec
>>> rare = make_label_dec(['slow','hard'],
... "Mix labels 'slow' and 'hard' for rare tests")
>>> @rare
... def f(): pass
...
>>>
>>> f.slow
True
>>> f.hard
True
```

needs_review(*msg*)

Skip a test that needs further review.

Parameters **msg** : string

msg regarding the review that needs to be done

CHAPTER

NINETYTHREE

TESTING.NITEST

93.1 Module: `testing.nitest`

Nipy Test Suite Runner.

The purpose of this module is to get the same behaviour on the command line as we do when do the following from ipython:

```
import nipy as ni ni.test()
```

We need to register the nose plugins defined in numpy. Currently we're only registering the KnownFailure plugin so the output is suppressed.

Copied and slightly modified from Fernando's IPython iptest.py module.

main()
Run NIPY test suite.

CHAPTER

NINETYFOUR

UTILS.GET_DATA

94.1 Module: `utils.get_data`

Download the nipy test/example data from the cirl.berkeley.edu server.

This utility asks the user if they would like to download the file and if so it:

- makes the data directory ~/.nipy/tests/data
- downloads the tarball
- extracts the tarball

94.2 Functions

`extract_tarfile`(*filename, dstdir*)
Extract tarfile to the destination directory.

`get_data`()

`read_chunk`(*fp*)

CHAPTER

NINETYFIVE

UTILS.MLABTEMP

95.1 Module: utils.mlabtemp

Create matlab-compatible temporary files.

mlab_tempfile(*dir=None*)

Returns a temporary file-like object with valid matlab name.

The file name is accessible as the .name attribute of the returned object. The caller is responsible for closing the returned object, at which time the underlying file gets deleted from the filesystem.

Parameters **dir** : str

A path to use as the starting directory. Note that this directory must already exist, it is NOT created if it doesn't (in that case, OSError is raised instead).

Returns **f** : A file-like object.

Examples

```
>>> f = mlab_tempfile()
>>> '-' not in f.name
True
>>> f.close()
```

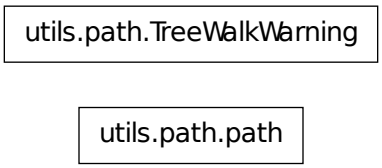
CHAPTER

NINETYSIX

UTILS.PATH

96.1 Module: utils.path

Inheritance diagram for nipy.utils.path:



path.py - An object representing a path to a file or directory.

Example:

from nipy.utils.path import path d = path('/home/guido/bin') for f in d.files('*.py'):

f.chmod(0755)

This module requires Python 2.2 or later.

URL: http://www.jorendorff.com/articles/python/path Author: Jason Orendorff <jason.orendorff@gmail.com> (and others - see the url!) Date: 7 Mar 2004

96.2 Classes

96.2.1 TreeWalkWarning

class TreeWalkWarning()

Bases: exceptions.Warning

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

96.2.2 path

class path()
Bases: str

Represents a filesystem path.

For documentation on individual methods, consult their counterparts in os.path.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

abspath()

access(*mode*)
Return true if current user has access to this path.

mode - One of the constants os.F_OK, os.R_OK, os.W_OK, os.X_OK

atime
Last access time of the file.

basename(*p*)
Returns the final component of a pathname

bytes()
Open this file, read all bytes, return them as a string.

chmod(*mode*)

chown(*uid, gid*)

chroot()

copy(*src, dst*)
Copy data and mode bits ("cp src dst").

The destination may be a directory.

copy2(*src, dst*)
Copy data and all stat info ("cp -p src dst").

The destination may be a directory.

copyfile(*src, dst*)
Copy data from src to dst

copymode(*src, dst*)
Copy mode bits from src to dst

copystat(*src, dst*)
Copy all stat info (mode bits, atime, mtime, flags) from src to dst

copytree(*src, dst, symlinks=False, ignore=None*)
Recursively copy a directory tree using copy2().

The destination directory must not already exist. If exception(s) occur, an Error is raised with a list of reasons.

If the optional symlinks flag is true, symbolic links in the source tree result in symbolic links in the destination tree; if it is false, the contents of the files pointed to by symbolic links are copied.

The optional ignore argument is a callable. If given, it is called with the *src* parameter, which is the directory being visited by copytree(), and *names* which is the list of *src* contents, as returned by os.listdir():

callable(src, names) -> ignored_names

Since copytree() is called recursively, the callable will be called once for each directory that is copied. It returns a list of names relative to the *src* directory that should not be copied.

XXX Consider this example code rather than the ultimate tool.

ctime
Creation time of the file.

dirname()

dirs(*pattern=None*)
D.dirs() -> List of this directory's subdirectories.

The elements of the list are path objects. This does not walk recursively into subdirectories (but see path.walkdirs).

With the optional 'pattern' argument, this only lists directories whose names match the given pattern. For example, d.dirs('build-*').

drive
The drive specifier, for example 'C:'. This is always empty on systems that don't use drive specifiers.

exists(*path*)
Test whether a path exists. Returns False for broken symbolic links

expand()
Clean up a filename by calling expandvars(), expanduser(), and normpath() on it.

This is commonly everything needed to clean up a filename read from a configuration file, for example.

expanduser()

expandvars()

ext
The file extension, for example '.py'.

files(*pattern=None*)
D.files() -> List of the files in this directory.

The elements of the list are path objects. This does not walk into subdirectories (see path.walkfiles).

With the optional 'pattern' argument, this only lists files whose names match the given pattern. For example, d.files('*.pyc').

fnmatch(*pattern*)
Return True if self.name matches the given pattern.

pattern - A filename pattern with wildcards, for example '*.py'.

get_owner()
Return the name of the owner of this file or directory.

This follows symbolic links.

On Windows, this returns a name of the form ur'DOMAINUser Name'. On Windows, a group can own a file or directory.

getatime(*filename*)
Return the last access time of a file, reported by os.stat().

getctime(*filename*)
Return the metadata change time of a file, reported by os.stat().

class **getcwd**()
Return the current working directory as a path object.

getmtime (*filename*)
Return the last modification time of a file, reported by os.stat().

getsize (*filename*)
Return the size of a file, reported by os.stat().

glob (*pattern*)
Return a list of path objects that match the pattern.

pattern - a path relative to this directory, with wildcards.

For example, path('/users').glob('*/bin/*') returns a list of all the files users have in their bin directories.

isabs (*s*)
Test whether a path is absolute

isdir (*s*)
Return true if the pathname refers to an existing directory.

isfile (*path*)
Test whether a path is a regular file

islink (*path*)
Test whether a path is a symbolic link

ismount (*path*)
Test whether a path is a mount point

joinpath (**args*)
Join two or more path components, adding a separator character (os.sep) if needed. Returns a new path object.

lines (*encoding=None, errors='strict', retain=True*)
Open this file, read all lines, return them in a list.

Optional arguments: encoding - The Unicode encoding (or character set) of the file. The default is None, meaning the content of the file is read as 8-bit characters and returned as a list of (non-Unicode) str objects.

errors - How to handle Unicode errors; see help(str.decode) for the options. Default is 'strict'

retain - If true, retain newline characters; but all newline character combinations ('r', 'n', 'rn') are translated to 'n'. If false, newline characters are stripped off. Default is True.

This uses 'U' mode in Python 2.3 and later.

link (*newpath*)
Create a hard link at 'newpath', pointing to this file.

listdir (*pattern=None*)
D.listdir() -> List of items in this directory.

Use D.files() or D.dirs() instead if you want a listing of just files or just subdirectories.

The elements of the list are path objects.

With the optional 'pattern' argument, this only lists items whose names match the given pattern.

lstat ()
Like path.stat(), but do not follow symbolic links.

makedirs (*mode=511*)

mkdir (*mode=511*)

move(*src, dst*)
Recursively move a file or directory to another location. This is similar to the Unix "mv" command.

If the destination is a directory or a symlink to a directory, the source is moved inside the directory. The destination path must not already exist.

If the destination already exists but is not a directory, it may be overwritten depending on os.rename() semantics.

If the destination is on our current filesystem, then rename() is used. Otherwise, src is copied to the destination and then removed. A lot more could be done here... A look at a mv.c shows a lot of the issues this implementation glosses over.

mtime
Last-modified time of the file.

name
The name of this file or directory without the full path.

For example, path('/usr/local/lib/libpython.so').name == 'libpython.so'

namebase
The same as path.name, but with one file extension stripped off.

For example, path('/home/guido/python.tar.gz').name == 'python.tar.gz', but path('/home/guido/python.tar.gz').namebase == 'python.tar'

normcase()

normpath()

open(*mode='r'*)
Open this file. Return a file object.

owner
Name of the owner of this file or directory.

parent
This path's parent directory, as a new path object.

For example, path('/usr/local/lib/libpython.so').parent == path('/usr/local/lib')

pathconf(*name*)

read_md5()
Calculate the md5 hash for this file.

This reads through the entire file.

readlink()
Return the path to which this symbolic link points.

The result may be an absolute or a relative path.

readlinkabs()
Return the path to which this symbolic link points.

The result is always an absolute path.

realpath()

relpath()
Return this path as a relative path, based from the current working directory.

relpathto(*dest*)
Return a relative path from self to dest.

If there is no relative path from self to dest, for example if they reside on different drives in Windows, then this returns dest.abspath().

remove()

removedirs()

rename(*new*)

renames(*new*)

rmdir()

rmtree(*path, ignore_errors=False, onerror=None*)
Recursively delete a directory tree.

If ignore_errors is set, errors are ignored; otherwise, if onerror is set, it is called to handle the error with arguments (func, path, exc_info) where func is os.listdir, os.remove, or os.rmdir; path is the argument to that function that caused it to fail; and exc_info is a tuple returned by sys.exc_info(). If ignore_errors is false and onerror is None, an exception is raised.

samefile(*f1, f2*)
Test whether two pathnames reference the same actual file

size
Size of the file, in bytes.

splitall()
Return a list of the path components in this path.

The first item in the list will be a path. Its value will be either os.curdir, os.pardir, empty, or the root directory of this path (for example, '/' or 'C:\'). The other items in the list will be strings.

path.path.joinpath(*result) will yield the original path.

splitdrive()
p.splitdrive() -> Return (p.drive, <the rest of p>).

Split the drive specifier from this path. If there is no drive specifier, p.drive is empty, so the return value is simply (path(''), p). This is always the case on Unix.

splitext()
p.splitext() -> Return (p.stripext(), p.ext).

Split the filename extension from this path and return the two parts. Either part may be empty.

The extension is everything from '.' to the end of the last path segment. This has the property that if (a, b) == p.splitext(), then a + b == p.

splitpath()
p.splitpath() -> Return (p.parent, p.name).

stat()
Perform a stat() system call on this path.

statvfs()
Perform a statvfs() system call on this path.

stripext()
p.stripext() -> Remove one file extension from the path.

For example, path('/home/guido/python.tar.gz').stripext() returns path('/home/guido/python.tar').

symlink (*newlink*)
Create a symbolic link at 'newlink', pointing here.

text (*encoding=None, errors='strict'*)
Open this file, read it in, return the content as a string.

This uses 'U' mode in Python 2.3 and later, so 'rn' and 'r' are automatically translated to 'n'.

Optional arguments:

encoding - The Unicode encoding (or character set) of the file. If present, the content of the file is decoded and returned as a unicode object; otherwise it is returned as an 8-bit str.

errors - How to handle Unicode errors; see help(str.decode) for the options. Default is 'strict'.

touch ()
Set the access/modified times of this file to the current time. Create the file if it does not exist.

unlink ()

utime (*times*)
Set the access and modified times of this file.

walk (*pattern=None, errors='strict'*)
D.walk() -> iterator over files and subdirs, recursively.

The iterator yields path objects naming each child item of this directory and its descendants. This requires that D.isdir().

This performs a depth-first traversal of the directory tree. Each directory is returned just before all its children.

The errors= keyword argument controls behavior when an error occurs. The default is 'strict', which causes an exception. The other allowed values are 'warn', which reports the error via warnings.warn(), and 'ignore'.

walkdirs (*pattern=None, errors='strict'*)
D.walkdirs() -> iterator over subdirs, recursively.

With the optional 'pattern' argument, this yields only directories whose names match the given pattern. For example, mydir.walkdirs('*test') yields only directories with names ending in 'test'.

The errors= keyword argument controls behavior when an error occurs. The default is 'strict', which causes an exception. The other allowed values are 'warn', which reports the error via warnings.warn(), and 'ignore'.

walkfiles (*pattern=None, errors='strict'*)
D.walkfiles() -> iterator over files in D, recursively.

The optional argument, pattern, limits the results to files with names that match the pattern. For example, mydir.walkfiles('*.tmp') yields only files with the .tmp extension.

write_bytes (*bytes, append=False*)
Open this file and write the given bytes to it.

Default behavior is to overwrite any existing file. Call p.write_bytes(bytes, append=True) to append instead.

write_lines (*lines, encoding=None, errors='strict', linesep='n', append=False*)
Write the given lines of text to this file.

By default this overwrites any existing file at this path.

This puts a platform-specific newline sequence on every line. See 'linesep' below.

lines - A list of strings.

encoding - A Unicode encoding to use. This applies only if 'lines' contains any Unicode strings.

errors - How to handle errors in Unicode encoding. This also applies only to Unicode strings.

linesep - The desired line-ending. This line-ending is applied to every line. If a line already has any standard line ending ('r', 'n', 'rn', u'x85', u'rx85', u'u2028'), that will be stripped off and this will be used instead. The default is os.linesep, which is platform-dependent ('rn' on Windows, 'n' on Unix, etc.) Specify None to write the lines as-is, like file.writelines().

Use the keyword argument append=True to append lines to the file. The default is to overwrite the file. Warning: When you use this with Unicode data, if the encoding of the existing data in the file is different from the encoding you specify with the encoding= parameter, the result is mixed-encoding data, which can really confuse someone trying to read the file later.

write_text (*text, encoding=None, errors='strict', linesep='n', append=False*)
Write the given text to this file.

The default behavior is to overwrite any existing file; to append instead, use the 'append=True' keyword argument.

There are two differences between path.write_text() and path.write_bytes(): newline handling and Unicode handling. See below.

Parameters:

- text - str/unicode - The text to be written.
- encoding - str - The Unicode encoding that will be used. This is ignored if 'text' isn't a Unicode string.
- errors - str - How to handle Unicode encoding errors. Default is 'strict'. See help(unicode.encode) for the options. This is ignored if 'text' isn't a Unicode string.
- linesep - keyword argument - str/unicode - The sequence of characters to be used to mark end-of-line. The default is os.linesep. You can also specify None; this means to leave all newlines as they are in 'text'.
- append - keyword argument - bool - Specifies what to do if the file already exists (True: append to the end of it; False: overwrite it.) The default is False.

— Newline handling.

write_text() converts all standard end-of-line sequences ('n', 'r', and 'rn') to your platform's default end-of-line sequence (see os.linesep; on Windows, for example, the end-of-line marker is 'rn').

If you don't like your platform's default, you can override it using the 'linesep=' keyword argument. If you specifically want write_text() to preserve the newlines as-is, use 'linesep=None'.

This applies to Unicode text the same as to 8-bit text, except there are three additional standard Unicode end-of-line sequences: u'x85', u'rx85', and u'u2028'.

(This is slightly different from when you open a file for writing with fopen(filename, "w") in C or file(filename, 'w') in Python.)

— Unicode

If 'text' isn't Unicode, then apart from newline handling, the bytes are written verbatim to the file. The 'encoding' and 'errors' arguments are not used and must be omitted.

If 'text' is Unicode, it is first converted to bytes using the specified 'encoding' (or the default encoding if 'encoding' isn't specified). The 'errors' argument applies only to this conversion.

CHAPTER

NINETYSEVEN

UTILS.PERLPIE

97.1 Module: `utils.perlpie`

Utility function to perform global search and replace in a source tree.

perlpie will perform a global search and replace on all files in a directory recursively. It's a small python wrapper around the *perl -p -i -e* functionality, but with easier syntax. I **strongly recommend** running *perlpie* on files under source control. In this way it's easy to track your changes and if (more likely when) you discover your regular expression was wrong you can easily revert. I also recommend using *grin* to test your regular expressions before running *perlpie*.

97.1.1 Parameters

oldstring [regular expression] Regular expression matching the string you want to replace

newstring [string] The string you would like to replace the oldstring with. Note this is not a regular expression but the exact string. One exception to this rule is the at symbol @. This has special meaning in perl, so you need an escape character for this. See Examples below.

97.1.2 Requires

perl : The underlying language we're using to perform the search and replace.

grin : Grin is a tool written by Robert Kern to wrap *grep* and *find* with python and easier command line options.

97.1.3 Example

Replace all occurences of foo with bar:

```
perlpie foo bar
```

Replace numpy.testing with nipy's testing framework:

```
perlpie 'from\s+numpy\.testing.*' 'from nipy.testing import *'
```

Replace all @slow decorators in my code with @dec.super_slow. Here we have to escape the @ symbol which has special meaning in perl:

```
perlpie '\@slow' '\@dec.super_slow'
```

Remove all occurences of importing make_doctest_suite:

```
perlpie 'from\snipy\.utils\.testutils.*make_doctest_suite'
```

97.2 Functions

check_deps()

main()

perl_dash_pie(*oldstr, newstr*)
Use perl to replace the oldstr with the newstr.

Examples

To replace all occurences of 'import numpy as N' with 'import numpy as np' >>> from nipy.utils import perlpie >>> perlpie.perl_dash_pie('imports+numpys+ass+N', 'import numpy as np') grind | xargs perl -pi -e 's/imports+numpys+ass+N/import numpy as np/g'

Part VI

Publications

CHAPTER

NINETYEIGHT

PEER-REVIEWED PUBLICATIONS

K. Jarrod Millman, M. Brett, "Analysis of Functional Magnetic Resonance Imaging in Python," Computing in Science and Engineering, vol. 9, no. 3, pp. 52-55, May/June, 2007.

CHAPTER

NINETYNINE

POSTERS

Taylor JE, Worsley K, Brett M, Cointepas Y, Hunter J, Millman KJ, Poline J-B, Perez F. “BrainPy: an open source environment for the analysis and visualization of human brain data.” Meeting of the Organization for Human Brain Mapping, 2005. See the *BrainPy HBM abstract*.

Part VII

NIPY License Information

SOFTWARE LICENSE

CHAPTER

ONE

DOCUMENTATION LICENSE

BIBLIOGRAPHY

[boehm1981] Boehm, Barry W. (1981) *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.

[Prechelt2000ECS] Prechelt, Lutz. 2000. An Empirical Comparison of Seven Programming Languages. *IEEE Computer* 33, 23–29.

[Walston1977MPM] Walston, C E, and C P Felix. 1977. A Method of Programming Measurement and Estimation. *IBM Syst J* 16, 54-73.

MODULE INDEX

N

INDEX

Symbols

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