Neuroimaging in Python Documentation

Release 0.6.1.dev1

Neuroimaging in Python team.

February 20, 2024

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 neuroimaging 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

2.1 Summary

- if you don't have it, install Python using the instructions below;
- if you don't have it, install Pip using the instructions below;
- if you don't have them, install NumPy >= 1.14 and Scipy >= 1.0 using the instructions below;
- install Nipy with something like:

  ```
  pip3 install --user nipy
  ```

Note: These instructions are for Python 3. If you are using Python 2.7, use `python2` instead of `python3` and `pip2` instead of `pip3`, for the commands below.

2.2 Details

2.2.1 Install Python, Pip, Numpy and Scipy

First install Python 3, then install the Python package installer Pip.

Install Python 3 on Linux

We recommend:

- `sudo apt-get install -y python3 python3-tk` (Debian, Ubuntu);
- `sudo dnf install -y python3 python3-tkinter` (Fedora).

These are the bare minimum installs. You will almost certainly want to install the development tools for Python to allow you to compile other Python packages:

- `sudo apt-get install -y python3-dev` (Debian, Ubuntu);
- `sudo dnf install -y python3-devel` (Fedora).

Now *Install Pip on Linux or macOS*.

Install Python 3 on macOS

We recommend you install Python 3.5 or later using Homebrew (http://brew.sh/):

```
brew install python3
```

Homebrew is an excellent all-round package manager for macOS that you can use to install many other free / open-source packages.

Now *Install Pip on Linux or macOS*.

Install Pip on Linux or macOS

Pip can install packages into your main system directories (a *system* install), or into your own user directories (a *user* install). We strongly recommend *user* installs.

To get ready for user installs, put the user local install `bin` directory on your user's executable program `PATH`. First find the location of the user `bin` directory with:

```
python3 -c 'import site; print(site.USER_BASE + "/bin")'
```

This will give you a result like `/home/your_username/.local/bin` (Linux) or `/Users/your_username/Library/Python/3.5/bin` (macOS).

Use your favorite text editor to open the `~/.bashrc` file (Linux) or `.bash_profile` (macOSX) in your home directory.

Add these lines to end of the file:

```
# Put the path to the local bin directory into a variable
py3_local_bin=$(python3 -c 'import site; print(site.USER_BASE + "/bin")')
# Put the directory at the front of the system PATH
export PATH="$py3_local_bin:$PATH"
```

Save the file, and restart your terminal to load the configuration from your `~/.bashrc` (Linux) or `~/.bash_profile` (macOS) file. Confirm that you have the user install directory in your PATH, with:

```
echo $PATH
```

Now install the Python package installer Pip into your user directories (see: install pip with get-pip.py):

```
# Download the get-pip.py installer
curl -LO https://bootstrap.pypa.io/get-pip.py
# Execute the installer for Python 3 and a user install
python3 get-pip.py --user
```

Check you have the right version of the `pip3` command with:

```
which pip3
```

This should give you something like `/home/your_username/.local/bin/pip3` (Linux) or `/Users/your_username/Library/Python/3.5/bin` (macOS).

Now *Install Python 3, Pip, NumPy and Scipy on Windows*.

Install Python 3, Pip, NumPy and Scipy on Windows

It's worth saying here that very few scientific Python developers use Windows, so if you're thinking of making the switch to Linux or macOS, now you have another reason to do that.

Option 1: Anaconda

If you are installing on Windows, you might want to use the Python 3 version of Anaconda. This is a large installer that will install many scientific Python packages, including NumPy and Scipy, as well as Python itself, and Pip, the package manager.

The machinery for the Anaconda bundle is not completely open-source, and is owned by a company, Continuum Analytics. If you would prefer to avoid using the Anaconda installer, you can also use the Python standard Pip installer.

Option 2: Standard install

If you don't have Python / Pip, we recommend the instructions here to install them. You can also install Python / Pip via the Python 3 installer from the https://python.org website.

If you already have an old Python installation, you don't have Pip, and you don't want to upgrade, you will need to download and install Pip following the instructions at install pip with get-pip.py.

Now open a Cmd or Powershell terminal and run:

```
pip3 install --user numpy scipy
```

2.2.2 Install Nipy

Now you have Python and Pip:

```
pip3 install --user nipy
```

On Windows, macOS, and nearly all Linux versions on Intel, this will install a binary (Wheel) package of NiPy.

2.3 Other packages we recommend

- IPython: Interactive Python environment;
- Matplotlib: Python plotting library.

2.4 Building from latest source code

2.4.1 Dependencies for build

- A C compiler: Nipy does contain a few C extensions for optimized routines. Therefore, you must have a compiler to build from source. Use XCode for your C compiler on macOS. On Windows, you will need the Microsoft Visual C++ version corresponding to your Python version - see using MSVC with Python. On Linux you should have the packages you need after you install the `python3-dev` (Debian / Ubuntu) or `python3-devel` (Fedora) packages using the instructions above;

- Cython 0.12.1 or later: Cython is a language that is a fusion of Python and C. It allows us to write fast code using Python and C syntax, so that it is easier to read and maintain than C code with the same functionality;
- Git version control software: follow the instructions on the main git website to install Git on Linux, macOS or Windows.

2.4.2 Procedure

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

```
# install Cython
pip3 install --user cython
```

```
# Clone the project repository
git clone https://github.com/nipy/nipy
```

to get the latest development version, and:

```
# Build the latest version in-place
cd nipy
pip3 install --user --editable .
```

to install the code in the development tree into your Python path.

2.5 Installing useful data files

See data-files for some instructions on installing data packages.

CHAPTER

THREE

GEOGRAPHY OF THE SCIPY WORLD

in which we briefly describe the various components you are likely to come across when writing scientific python software in general, and NIPY code in particular.

3.1 Numpy

NumPy is the basic Python array-manipulation package. It allows you to create, slice and manipulate N-D arrays at near C speed. It also has basic arithmetical and mathematical functions (such as sum, mean, and log, exp, sin, cos), matrix multiplication (`numpy.dot`), Fourier transforms (`numpy.fft`) and basic linear algebra `numpy.linalg`.

3.2 SciPy

Scipy is a large umbrella project that builds on Numpy (and depends on it). It includes a variety of high level science and engineering modules together as a single package. There are extended modules for linear algebra (including wrappers to BLAS and LAPACK), optimization, integration, sparse matrices, special functions, FFTs, signal and image processing, genetic algorithms, ODE solvers, and others.

3.3 Matplotlib

Matplotlib is a 2D plotting package that depends on NumPy. It has a simple matlab-like plotting syntax that makes it relatively easy to create good-looking plots, histograms and images with a small amount of code. As well as this simplified Matlab-like syntax, There is also a more powerful and flexible object-oriented interface.

3.4 Ipython

Ipython is an interactive shell for python that has various features of the interactive shell of Matlab, Mathematica and R. It works particularly well with Matplotlib, but is also an essential tool for interactive code development and code exploration. It contains libraries for creainteracting with parallel jobs on clusters or over several CPU cores in a fairly transparent way.

3.5 Cython

Cython is a development language that allows you to write a combination of Python and C-like syntax to generate Python extensions. It is especially good for linking C libraries to Python in a readable way. It is also an excellent choice for optimization of Python code, because it allows you to drop down to C or C-like code at your bottlenecks without losing much of the readability of Python.

3.6 Mayavi

Mayavi is a high-level python interface to the VTK plotting libraries.

CHAPTER

FOUR

TUTORIALS

4.1 Basic Data IO

Accessing images using nipy:

While Nifti is the primary file format Analyze images (with associated .mat file), and MINC files can also be read.

4.1.1 Load Image from File

Get a filename for an example file. `anatfile` gives a filename for a small testing image in the nipy distribution:

```
>>> from nipy.testing import anatfile
```

Load the file from disk:

```
>>> from nipy import load_image
>>> myimg = load_image(anatfile)
>>> myimg.shape
(33, 41, 25)
>>> myimg.affine
array([[ -2.,   0.,   0.,  32.],
       [  0.,   2.,   0., -40.],
       [  0.,   0.,   2., -16.],
       [  0.,   0.,   0.,   1.]])
```

4.1.2 Access Data into an Array

This allows the user to access data as a numpy array.

```
>>> mydata = myimg.get_fdata()
>>> mydata.shape
(33, 41, 25)
>>> mydata.ndim
3
```

4.1.3 Save image to a File

```
>>> from nipy import save_image
>>> newimg = save_image(myimg, 'newmyfile.nii')
```

4.1.4 Create Image from an Array

This will have a generic affine-type CoordinateMap with unit voxel sizes.

```
>>> import numpy as np
>>> from nipy.core.api import Image, vox2mni
>>> rawarray = np.zeros((43,128,128))
>>> arr_img = Image(rawarray, vox2mni(np.eye(4)))
>>> arr_img.shape
(43, 128, 128)
```

4.1.5 Coordinate map

Images have a Coordinate Map.

The Coordinate Map contains information defining the input (domain) and output (range) Coordinate Systems of the image, and the mapping between the two Coordinate systems. The *input* coordinate system is the *voxel* coordinate system, and the *output* coordinate system is the *world* coordinate system.

```
>>> newimg.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k'), name='voxels', coord_
↪dtype=float64),
   function_range=CoordinateSystem(coord_names=('aligned-x=L->R', 'aligned-y=P->A',
↪'aligned-z=I->S'), name='aligned', coord_dtype=float64),
   affine=array([[ -2.,   0.,   0.,  32.],
                 [  0.,   2.,   0., -40.],
                 [  0.,   0.,   2., -16.],
                 [  0.,   0.,   0.,   1.]])
)
```

See *Basics of the Coordinate Map* for more detail.

4.2 Basics of the Coordinate Map

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

Coordinate Map

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

The *input* or *domain* in an image are voxel coordinates in the image array. The *output* or *range* are the millimetre coordinates in some space, that correspond to the input (voxel) coordinates.

```
>>> import nipy
```

Get a filename for an example file:

```
>>> from nipy.testing import anatfile
```

Get the coordinate map for the image:

```
>>> anat_img = nipy.load_image(anatfile)
>>> coordmap = anat_img.coordmap
```

For more on Coordinate Systems and their properties *nipy.core.reference.coordinate_system*

You can inspect a coordinate map:

```
>>> coordmap.function_domain.coord_names
>>> ('i', 'j', 'k')
```

```
>>> coordmap.function_range.coord_names
('aligned-x=L->R', 'aligned-y=P->A', 'aligned-z=I->S')
```

```
>>> coordmap.function_domain.name
'voxels'
>>> coordmap.function_range.name
'aligned'
```

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 (i,j,k) to the millimeter space (x,y,z)

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

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. This is the same as saying that the mapping takes points in the mapping function *domain* and transforms them to points in the mapping function *range*.

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.

```
>>> coordmap.affine
array([[ -2.,   0.,   0.,  32.],
       [  0.,   2.,   0., -40.],
       [  0.,   0.,   2., -16.],
       [  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 (i,j,k) to (x,y,z):

```
>>> coordmap([1,2,3])
array([ 30., -36., -10.])
```

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

```
>>> coordmap.inverse()([30.,-36.,-10.])
array([1., 2., 3.])
```

We can see how this works if we just apply the affine ourselves using dot product.

Note: 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)

```
>>> coordmap.affine
array([[ -2.,   0.,   0.,  32.],
       [  0.,   2.,   0., -40.],
       [  0.,   0.,   2., -16.],
       [  0.,   0.,   0.,   1.]])
```

```
>>> import numpy as np
>>> np.dot(coordmap.affine, np.transpose([1,2,3,1]))
array([ 30., -36., -10.,   1.])
```

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

4.2.1 Use of the Coordinate Map for spatial normalization

The Coordinate Map can be used to describe the transformations needed to perform spatial normalization. Suppose we have an anatomical Image from one subject *subject_img* and we want to create an Image in a standard space like Tailarach space. An affine registration algorithm will produce a 4-by-4 matrix representing the affine transformation, *T*, that takes a point in the subject's coordinates *subject_world* to a point in Tailarach space *tailarach_world*. The subject's Image has its own Coordinate Map, *subject_cmap* and there is a Coordinate Map for Tailarach space which we will call *tailarach_cmap*.

Having found the transformation matrix *T*, the next step in spatial normalization is usually to resample the array of *subject_img* so that it has the same shape as some atlas *atlas_img*. Note that because it is an atlas Image, *tailarach_camp=atlas_img.coordmap*.

A resampling algorithm uses an interpolator which needs to know which voxel of *subject_img* corresponds to which voxel of *atlas_img*. This is therefore a function from *atlas_voxel* to *subject_voxel*.

This function, paired with the information that it is a map from atlas-voxel to subject-voxel is another example of a Coordinate Map. The code to do this might look something like the following:

```
>>> from nipy.testing import anatfile, funcfile
>>> from nipy.algorithms.registration import HistogramRegistration
>>> from nipy.algorithms.kernel_smooth import LinearFilter
```

We'll make a smoothed version of the anatomical example image, and pretend it's the template

```
>>> smoother = LinearFilter(anat_img.coordmap, anat_img.shape)
>>> atlas_im = smoother.smooth(anat_img)
>>> subject_im = anat_img
```

We do an affine registration between the two.

```
>>> reggie = HistogramRegistration(subject_im, atlas_im)
>>> aff = reggie.optimize('affine').as_affine()
Initial guess...
...
```

Now we make a coordmap with this transformation

```
>>> from nipy.core.api import AffineTransform
>>> subject_cmap = subject_im.coordmap
>>> talairach_cmap = atlas_im.coordmap
>>> subject_world_to_talairach_world = AffineTransform(
...                                         subject_cmap.function_range,
...                                         talairach_cmap.function_range,
...                                         aff)
...
```

We resample the ‘subject’ image to the ‘atlas image

```
>>> from nipy.algorithms.resample import resample
>>> normalized_subject_im = resample(subject_im, talairach_cmap,
...                                  subject_world_to_talairach_world,
...                                  atlas_im.shape)
>>> normalized_subject_im.shape == atlas_im.shape
True
>>> normalized_subject_im.coordmap == atlas_im.coordmap
True
>>> np.all(normalized_subject_im.affine == atlas_im.affine)
True
```

4.2.2 Mathematical definition

For a more formal mathematical description of the coordinate map, see math-coordmap.

4.3 Specifying a GLM in NiPy

In this tutorial we will discuss NiPy’s model and specification of a fMRI experiment.

This involves:

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

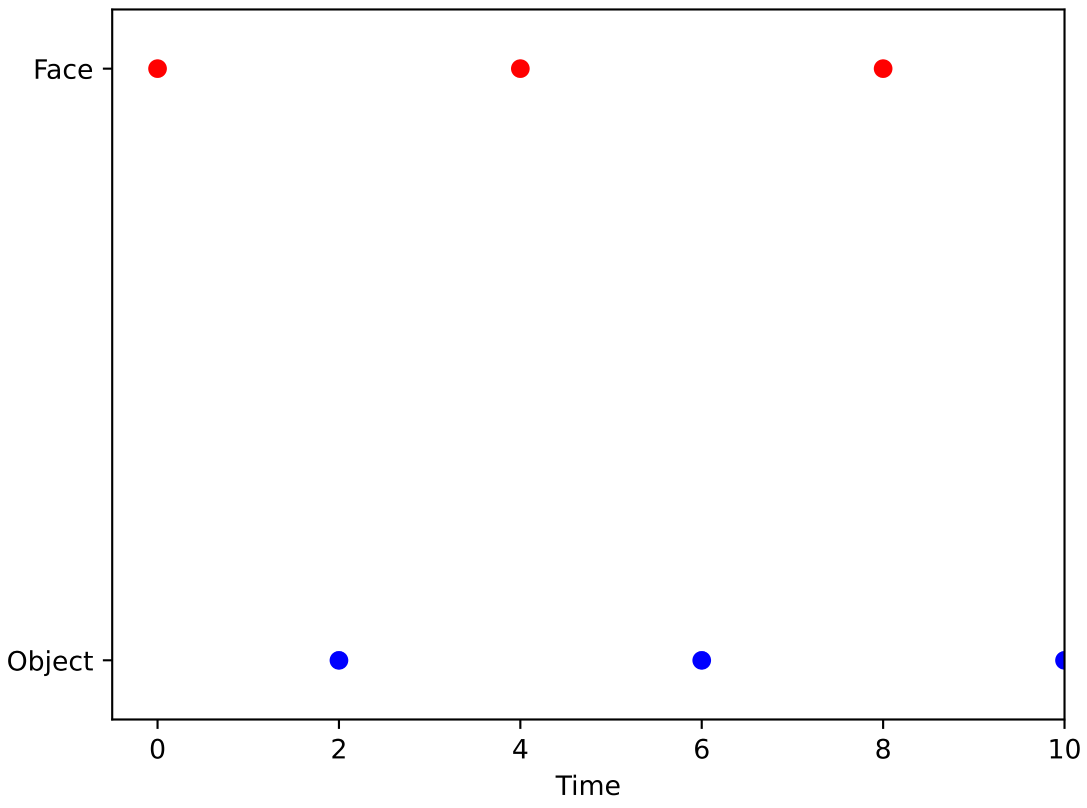
4.3.1 Experimental model

We first begin by describing typically encountered fMRI designs.

- Event-related categorical design, i.e. *Face* vs. *Object*
- Block categorical design
- Continuous stimuli, i.e. a rotating checkerboard
- Events with amplitudes, i.e. non-categorical values
- Events with random amplitudes

Event-related categorical design

This design is a canonical design in fMRI used, for instance, in an experiment designed to detect regions associated to discrimination between *Face* and *Object*. This design can be graphically represented in terms of delta-function responses that are effectively events of duration 0 and infinite height.



In this example, there *Face* event types are presented at times [0,4,8,12,16] and *Object* event types at times [2,6,10,14,18].

More generally, given a set of event types *V*, an event type experiment can be modeled as a sum of delta functions (point masses) at pairs of times and event types:

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

Formally, this can be thought of as realization of a marked point process, that says we observe 10 points in the space $\mathbb{R} \times V$ where V is the set of all event types. Alternatively, we can think of the experiment as a measure E on $\mathbb{R} \times V$

$$E([t_1, t_2] \times A) = \int_{t_1}^{t_2} \int_A dE(v, t)$$

This intensity measure determines, in words, "the amount of stimulus within *A* delivered in the interval $[t_1, t_2]$". In this categorical design, stimuli a_j are delivered as point masses at the times t_j.

Practically speaking, we can read this as saying that our experiment has 10 events, occurring at times $t_1, \dots, t_{10}$ with event types $a_1, \dots, a_{10} \in V$.

Typically, as in our *Face* vs *Object* example, the events occur in groups, say odd events are labelled a, even ones b. We might rewrite this as

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

This type of experiment can be represented by two counting processes, i.e. measures on $mathbbR$, (E_a, E_b) defined as

$$\begin{aligned} E_a(t) &= \sum_{t_j, j \text{ odd}} 1_{(-\infty,t_j]}(t) \\ &= E((-\infty,t],\{a\}) \\ E_b(t) &= \sum_{t_j, j \text{ even}} 1_{(-\infty,t_j]}(t) \\ &= E((-\infty,t],\{b\}) \end{aligned}$$

Counting processes vs. intensities

Though the experiment above can be represented in terms of the pair $(E_a(t), E_b(t))$, it is more common in neuroimaging applications to work with instantaneous intensities rather then cumulative intensities.

$$\begin{aligned} e_a(t) &= \frac{\partial}{\partial t} E_a(t) \\ e_b(t) &= \frac{\partial}{\partial t} E_b(t) \end{aligned}$$

For the time being, we will stick with cumulative intensities because it unifies the designs above. When we turn to the neuronal model below, we will return to the intensity model.

Block categorical design

For block designs of the *Face* vs. *Object* type, we might also allow event durations, meaning that we show the subjects a *Face* for a period of, say, 0.5 seconds. We might represent this experiment graphically as follows,

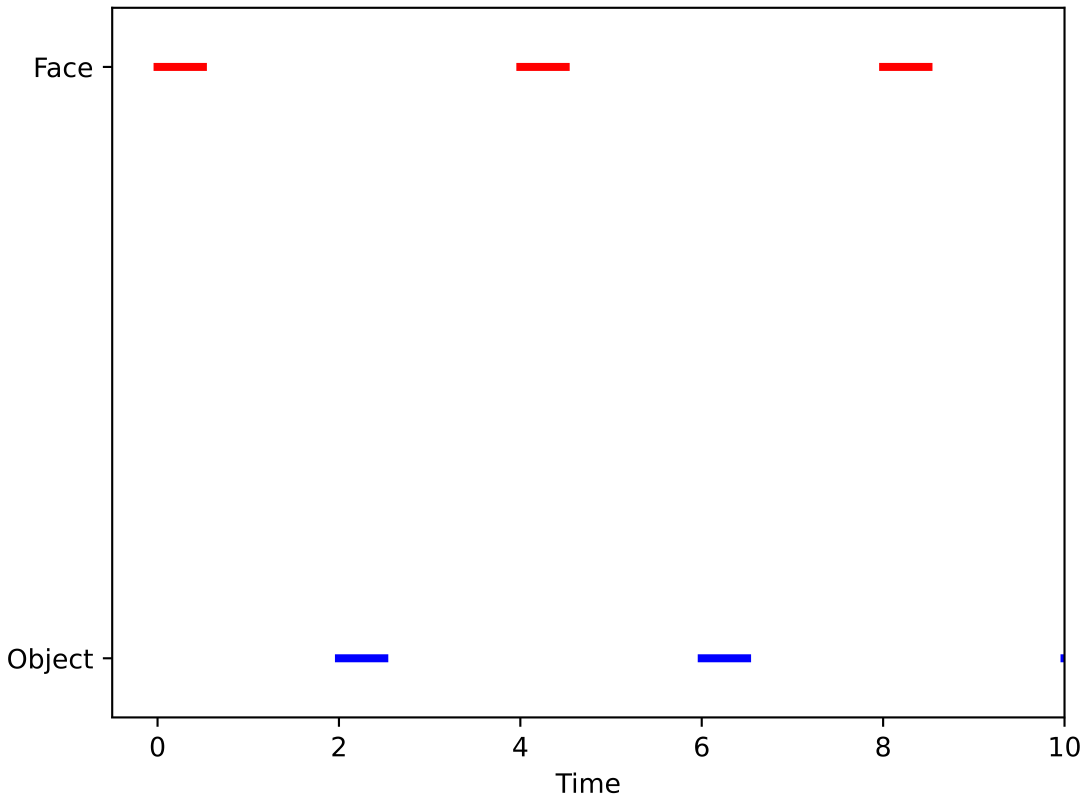
and the intensity measure for the experiment could be expressed in terms of

$$\begin{aligned} E_a(t) = E((-\infty,t],\{a\}) &= \sum_{t_j, j \text{ odd}} \frac{1}{0.5} \int_{t_j}^{\min(t_j+0.5,t)} ds \\ E_b(t) = E((-\infty,t],\{b\}) &= \sum_{t_j, j \text{ even}} \frac{1}{0.5} \int_{t_j}^{\min(t_j+0.5,t)} ds \end{aligned}$$

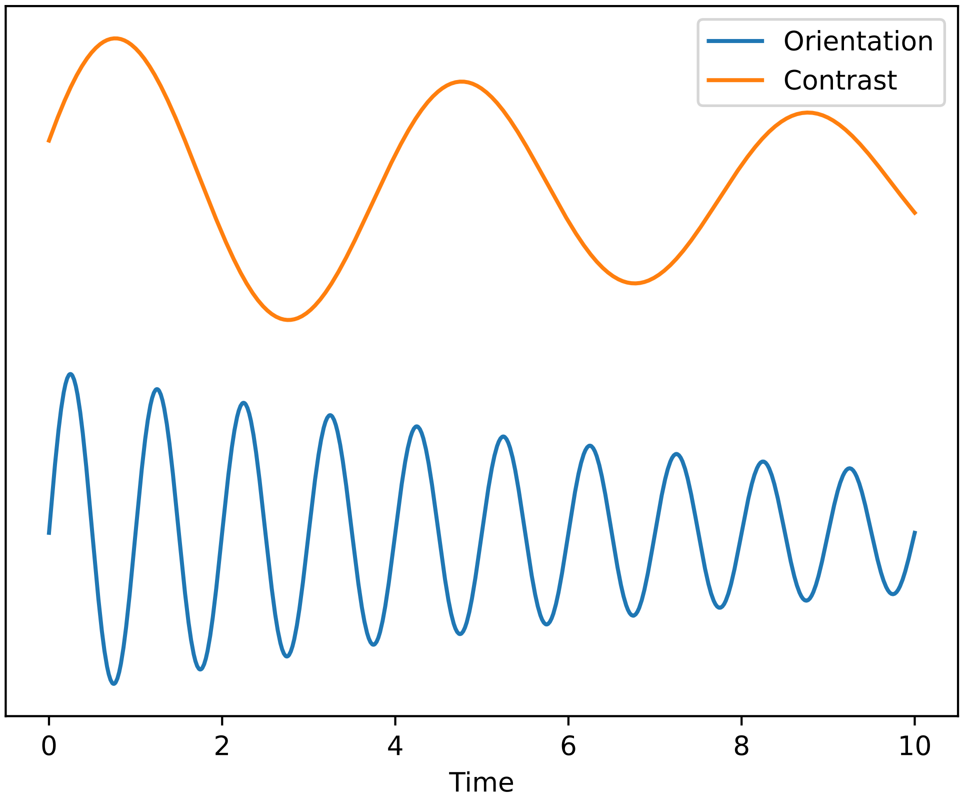
The normalization chosen above ensures that each event has integral 1, that is a total of 1 “stimulus unit” is presented for each 0.5 second block. This may or may not be desirable, and could easily be changed.

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 $(O(t), C(t))$. In this example we have set


Face
Object
0
2
4
6
8
10
Time

```
import numpy as np

t = np.linspace(0,10,1000)
o = np.sin(2*np.pi*(t+1)) * np.exp(-t/10)
c = np.sin(2*np.pi*(t+0.2)/4) * np.exp(-t/12)
```



The cumulative intensity measure for such an experiment might look like

$$E([t_1, t_2], A) = \int_{t_1}^{t_2} \left(\int_A \, dc \; do \right) \; dt.$$

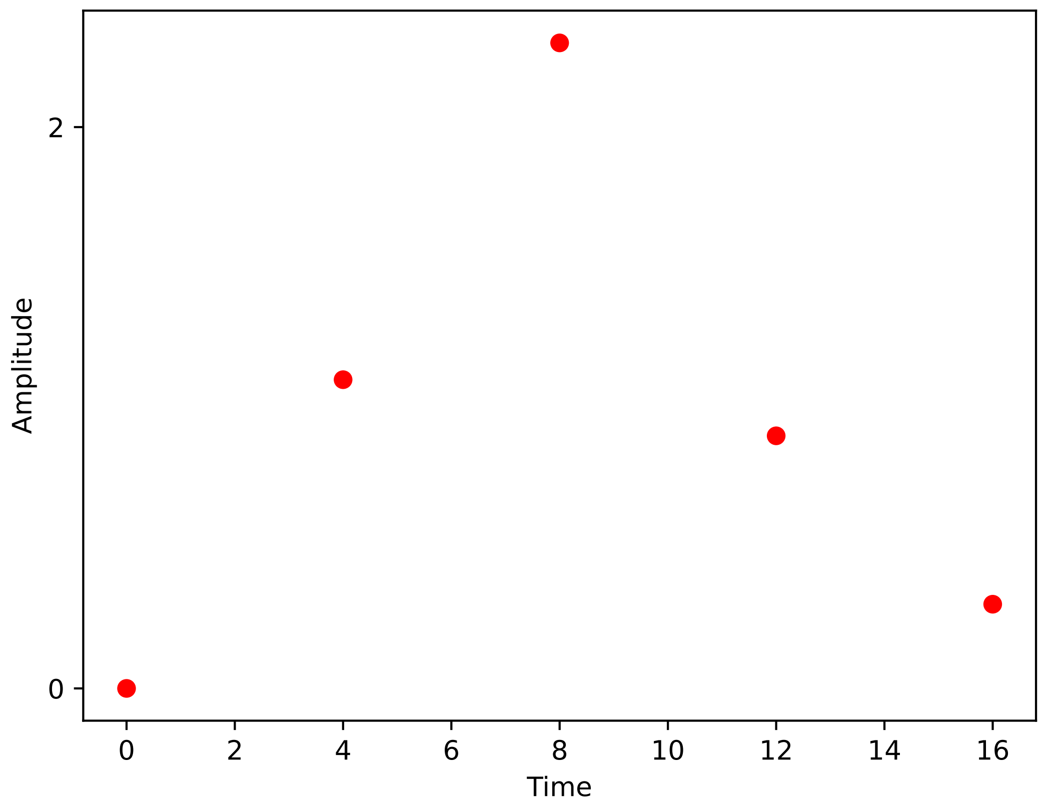
In words, this reads as $E([t_1, t_2], A)$ is the amount of time in the interval $[t_1, t_2]$ for which the state vector $(O(t), C(t))$ was in the region A.

Events with amplitudes

Another (event-related) experimental paradigm is one in which the event types have amplitudes, perhaps in a pain experiment with a heat stimulus, we might consider the temperature an amplitude. These amplitudes could be multi-valued. We might represent this parametric design mathematically as

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

which is virtually identical to our description of the *Face* vs. *Object* experiment in face-object though the values a_j are floats rather than labels. Graphically, this experiment might be represented as in this figure below.

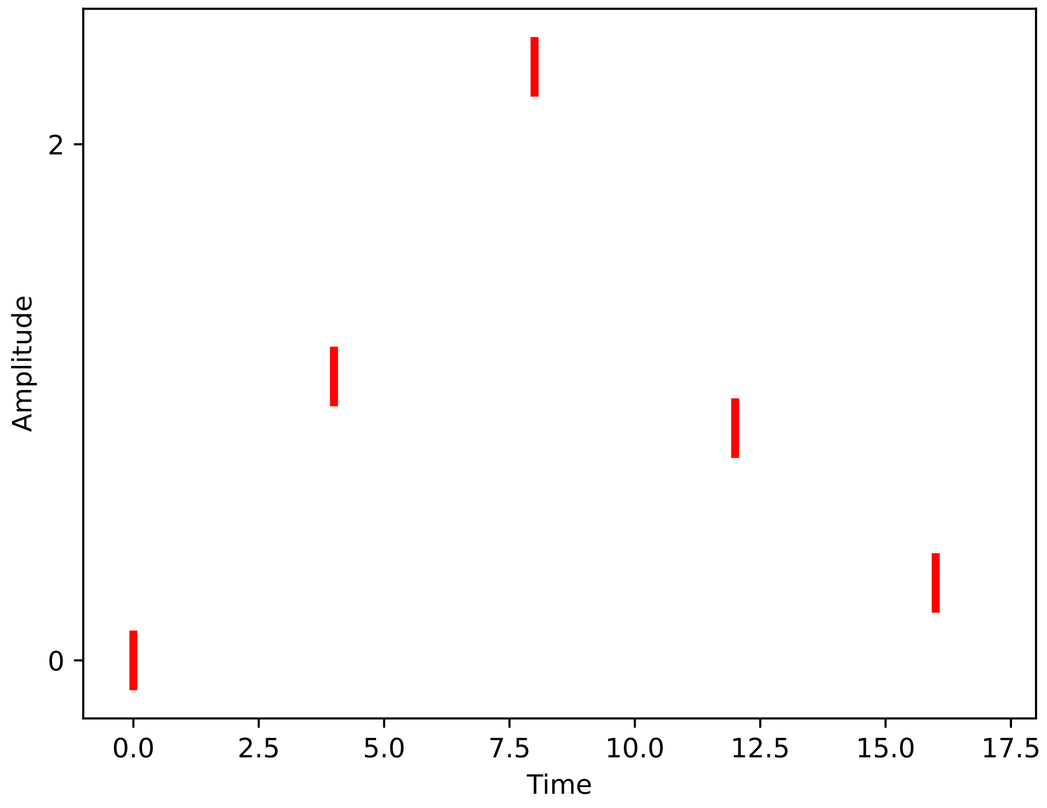


Events with random amplitudes

Another possible approach to specifying an experiment might be to deliver a randomly generated stimulus, say, uniformly distributed on some interval, at a set of prespecified event times.

We might represent this graphically as in the following figure.

Of course, the stimuli need not be randomly distributed over some interval, they could have fairly arbitrary distributions. Or, in the *Face* vs *Object* scenario, we could randomly present of one of the two types and the distribution at a particular event time t_j would be represented by a probability P_j.

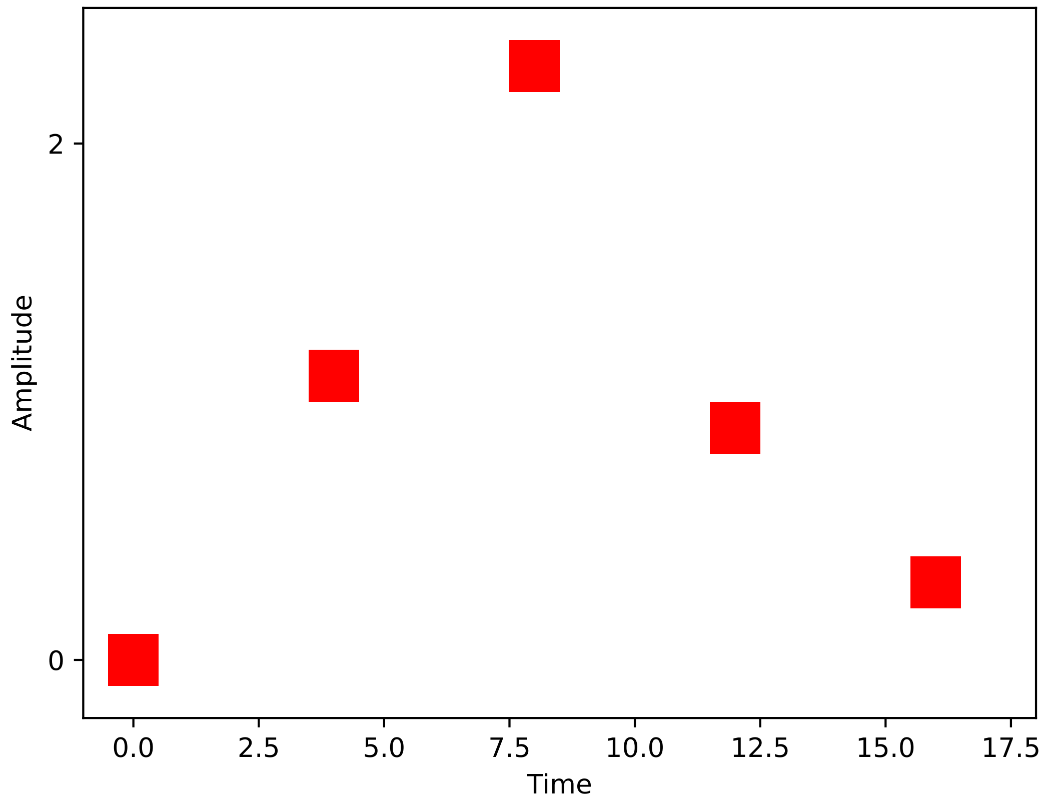

Amplitude
Time
0
2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

The cumulative intensity model for such an experiment might be

$$E([t_1,t_2],A) = \sum_j 1_{[t_1,t_2]}(t_j) \int_A P_j(da)$$

If the times were not prespecified but were themselves random, say uniform over intervals $[u_j, v_j]$, we might modify the cumulative intensity to be

$$E([t_1,t_2],A) = \sum_j \int_{\max(u_j,t_1)}^{\min(v_j,t_2)} \int_A P_j(da)\; dt$$



4.4 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. It is most commonly expressed as some linear function of the experiment E. As with the experimental model, we prefer to start off by working with the cumulative neuronal activity, a measure on $\mathbb{R}$, though, ultimately we will work with the intensities in intensity.

Typically, the neuronal model with an experiment model E has the form

$$N([t_1,t_2]) = \int_{t_1}^{t_2} \int_V f(v,t)\; dE(v,t)$$

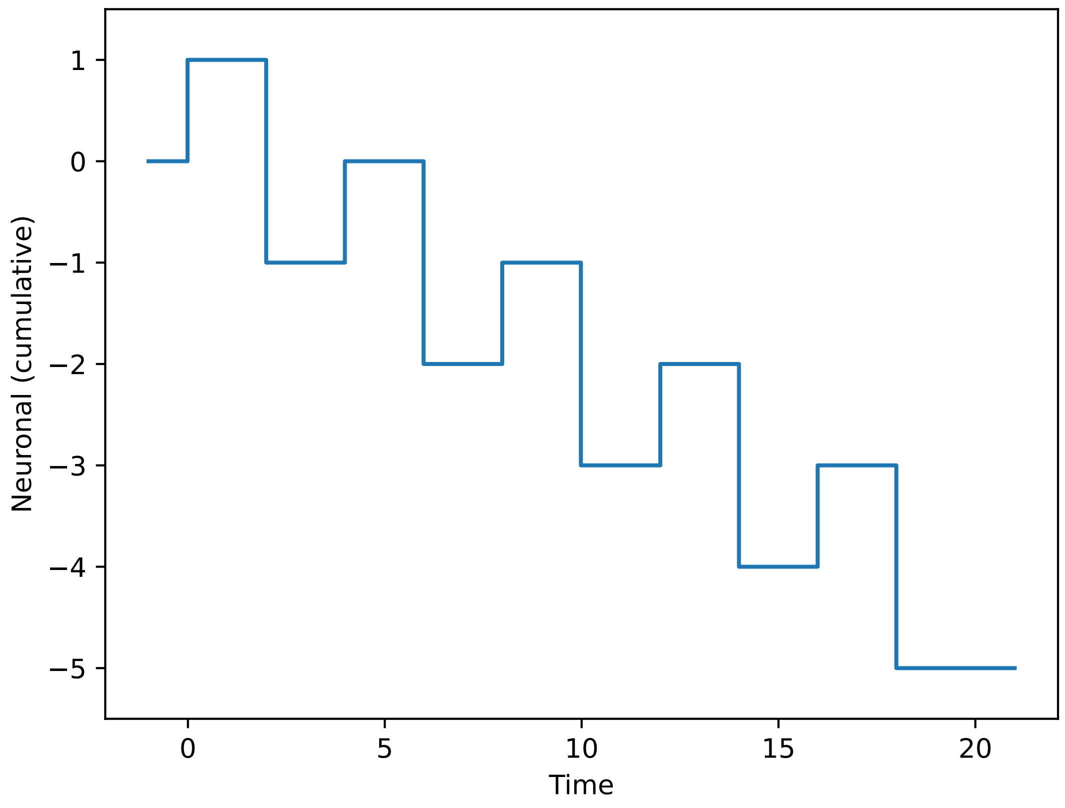
Unlike the experimental model, which can look somewhat abstract, the neuronal model can be directly modeled. For example, take the standard *Face* vs. *Object* model face-object, in which case $V = \{a, b\}$ and we can set

$$f(v,t) = \begin{cases} \beta_a & v = a \\ \beta_b & v = b \end{cases}$$

Thus, the cumulative neuronal model can be expressed as

```python
from sympy import Symbol, Heaviside
t = Symbol('t')
ta = [0,4,8,12,16]
tb = [2,6,10,14,18]
ba = Symbol('ba')
bb = Symbol('bb')
fa = sum([Heaviside(t-_t) for _t in ta]) * ba
fb = sum([Heaviside(t-_t) for _t in tb]) * bb
N = fa+fb
```

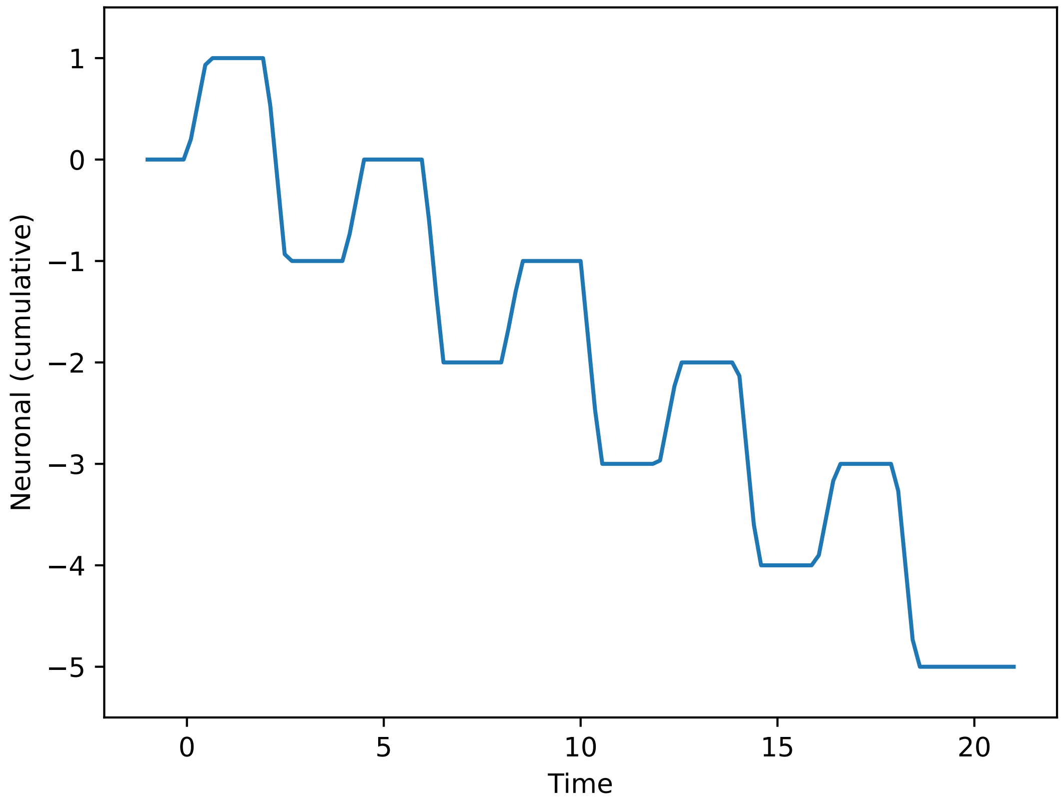
Or, graphically, if we set $\beta_a = 1$ and $\beta_b = -2$, as



In the block design, we might have the same form for the neuronal model (i.e. the same f above), but the different experimental model E yields

```
from sympy import Symbol, Piecewise
ta = [0,4,8,12,16]; tb = [2,6,10,14,18]
ba = Symbol('ba')
bb = Symbol('bb')
fa = sum([Piecewise((0, (t<_t)), ((t-_t)/0.5, (t<_t+0.5)), (1, (t >= _t+0.5))) for _t in␣
↪ta])*ba
fb = sum([Piecewise((0, (t<_t)), ((t-_t)/0.5, (t<_t+0.5)), (1, (t >= _t+0.5))) for _t in␣
↪tb])*bb
N = fa+fb
```

Or, graphically, if we set $\beta_a = 1$ and $\beta_b = -2$, as



The function f above can be expressed as

$$f(v,t) = \beta_a 1_{\{a\}}(v) + \beta_b 1_{\{b\}}(v) = \beta_a f_a(v,t) + \beta_b f_b(v,t)$$

Hence, our typical neuronal model can be expressed as a sum

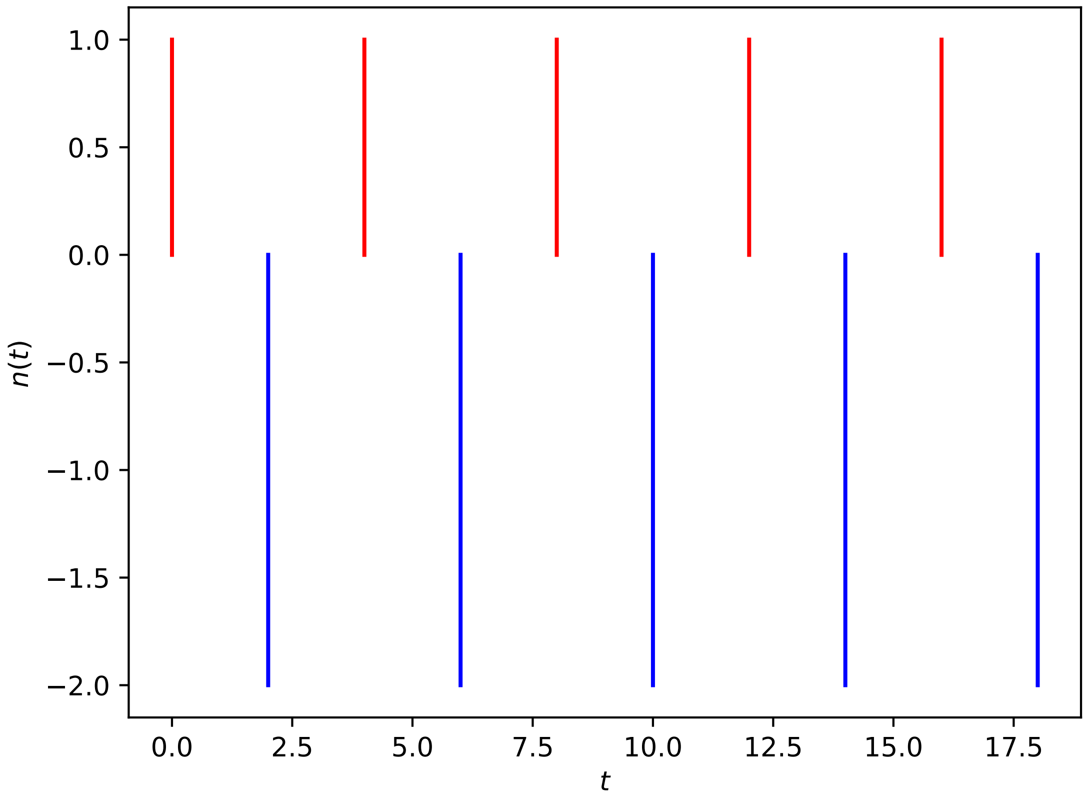
$$\begin{aligned} N([t_1,t_2]) &= \sum_i \beta_i \int_{t_1}^{t_2} \int_V f_i(v,t) \; dE(v,t) \\ &= \sum_i \beta_i \tilde{N}_{f_i}([t_1,t_2]) \end{aligned}$$

for arbitrary functions $\tilde{N}_{f_i}$. Above, $\tilde{N}_{f_i}$ represents the stimulus contributed to N from the function f_i. In the *Face* vs. *Object* example face-object, these cumulative intensities are related to the more common of neuronal model of intensities in terms of delta functions

$$\frac{\partial}{\partial t}\tilde{N}_{f_a}(t) = \beta_a \sum_{t_i : i \text{ odd}} \delta_{t_i}(t)$$

```
from sympy import Symbol, Heaviside
ta = [0,4,8,12,16]
t = Symbol('t')
ba = Symbol('ba')
fa = sum([Heaviside(t-_t) for _t in ta]) * ba
print(fa.diff(t))
```

```
ba*(DiracDelta(t) + DiracDelta(t - 16) + DiracDelta(t - 12) + DiracDelta(t - 8) +␣
↪DiracDelta(t - 4))
```



4.4.1 Convolution

In our continuous example above, with a periodic orientation and contrast, we might take

$$f_O(t,(o,c)) = o$$
$$f_O(t,(o,c)) = c$$

yielding a neuronal model

$$N([t_1,t_2]) = \beta_O O(t) + \beta_C C(t)$$

We might also want to allow a delay in the neuronal model

$$N^{\text{delay}}([t_1,t_2]) = \beta_O O(t-\tau_O) + \beta_C C(t-\tau_C).$$

This delay can be represented mathematically in terms of convolution (of measures)

$$N^{\text{delay}}([t_1,t_2]) = \left(\tilde{N}_{f_O} * \delta_{-\tau_O}\right)([t_1,t_2]) + \left(\tilde{N}_{f_C} * \delta_{-\tau_C}\right)([t_1,t_2])$$

Another model that uses convolution is the *Face* vs. *Object* one in which the neuronal signal is attenuated with an exponential decay at time scale τ

$$D([t_1,t_2]) = \int_{\max(t_1,0)}^{t_2} \tau e^{-\tau t}\, dt$$

yielding

$$N^{\text{decay}}([t_1,t_2]) = (N * D)[t_1,t_2]$$

4.5 Events with amplitudes

We described a model above event-amplitude with events that each have a continuous value a attached to them. In terms of a neuronal model, it seems reasonable to suppose that the (cumulative) neuronal activity is related to some function, perhaps expressed as a polynomial $h(a) = \sum_j \beta_j a^j$ yielding a neuronal model

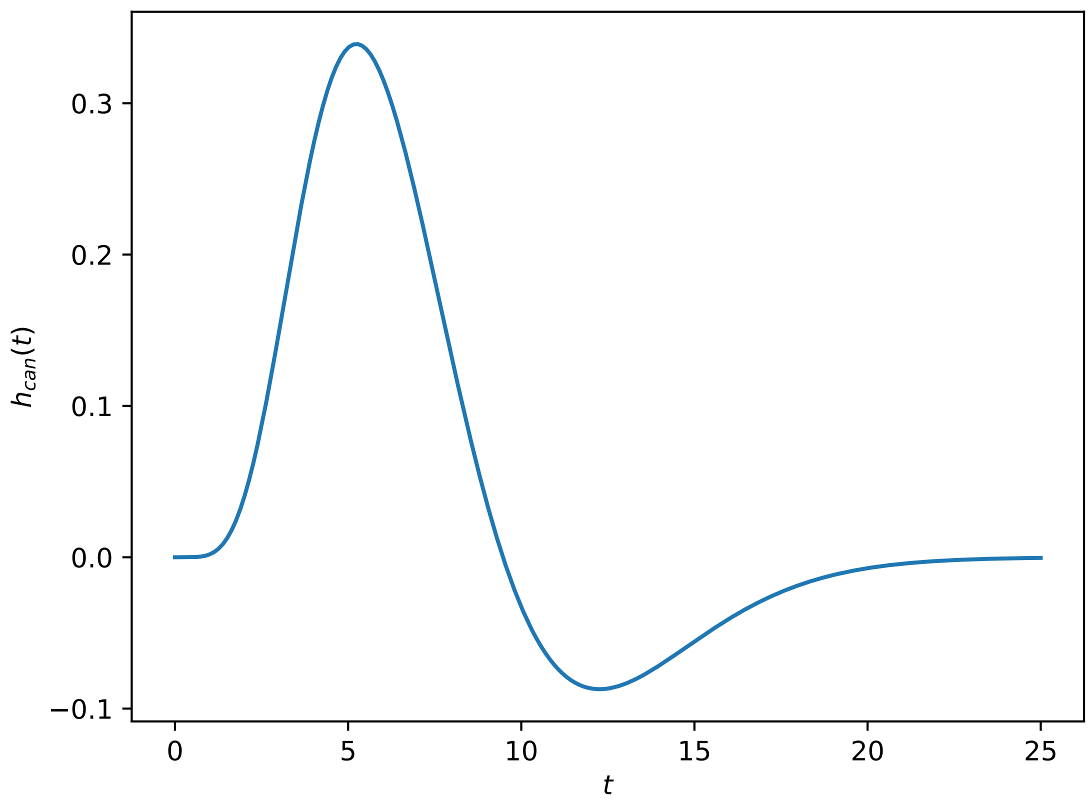
$$N([t_1,t_2]) = \sum_j \beta_j \tilde{N}_{a^j}([t_1,t_2])$$

4.5.1 Hemodynamic model

The hemodynamic model is a model for the BOLD signal, expressed as some function of the neuronal model. The most common hemodynamic model is just the convolution of the neuronal model with some hemodynamic response function, HRF

$$HRF((-\infty,t]) = \int_{-\infty}^{t} h_{can}(s)\, ds$$
$$H([t_1,t_2]) = (N * HRF)[t_1,t_2]$$

The canonical one is a difference of two Gamma densities



$h_{can}(t)$
0.3
0.2
0.1
0.0
−0.1
0
5
10
15
20
25
t

4.5.2 Intensities

Hemodynamic models are, as mentioned above, most commonly expressed in terms of instantaneous intensities rather than cumulative intensities. Define

$$n(t) = \frac{\partial}{\partial t} N((-\infty, t]).$$

The simple model above can then be written as

$$h(t) = \frac{\partial}{\partial t}(N * HRF)(t) = \int_{-\infty}^{\infty} n(t-s) h_{can}(s) \, ds.$$

In the *Face* vs. *Object* experiment, the integrals above can be evaluated explicitly because $n(t)$ is a sum of delta functions

$$n(t) = \beta_a \sum_{t_i : i \text{ odd}} \delta_{t_i}(t) + \beta_b \sum_{t_i : i \text{ even}} \delta_{t_i}(t)$$

In this experiment we may want to allow different hemodynamic response functions within each group, say h_a within group a and h_b within group b. This yields a hemodynamic model

$$h(t) = \beta_a \sum_{t_i : i \text{ odd}} h_a(t - t_i) + \beta_b \sum_{t_i : i \text{ even}} h_b(t - t_i)$$

```python
from nipy.modalities.fmri import hrf

ta = [0,4,8,12,16]; tb = [2,6,10,14,18]
ba = 1; bb = -2
na = ba * sum([hrf.glover(hrf.T - t) for t in ta])
nb = bb * sum([hrf.afni(hrf.T - t) for t in tb])
n = na + nb
```

Applying the simple model to the events with amplitude model and the canonical HRF yields a hemodynamic model

$$h(t) = \sum_{i,j} \beta_j a_i^j h_{can}(t - t_i)$$

```python
import numpy as np
from nipy.modalities.fmri.utils import events, Symbol

a = Symbol('a')
b = np.linspace(0,50,6)
amp = b*([-1,1]*3)
d = events(b, amplitudes=amp, g=a+0.5*a**2, f=hrf.glover)
```

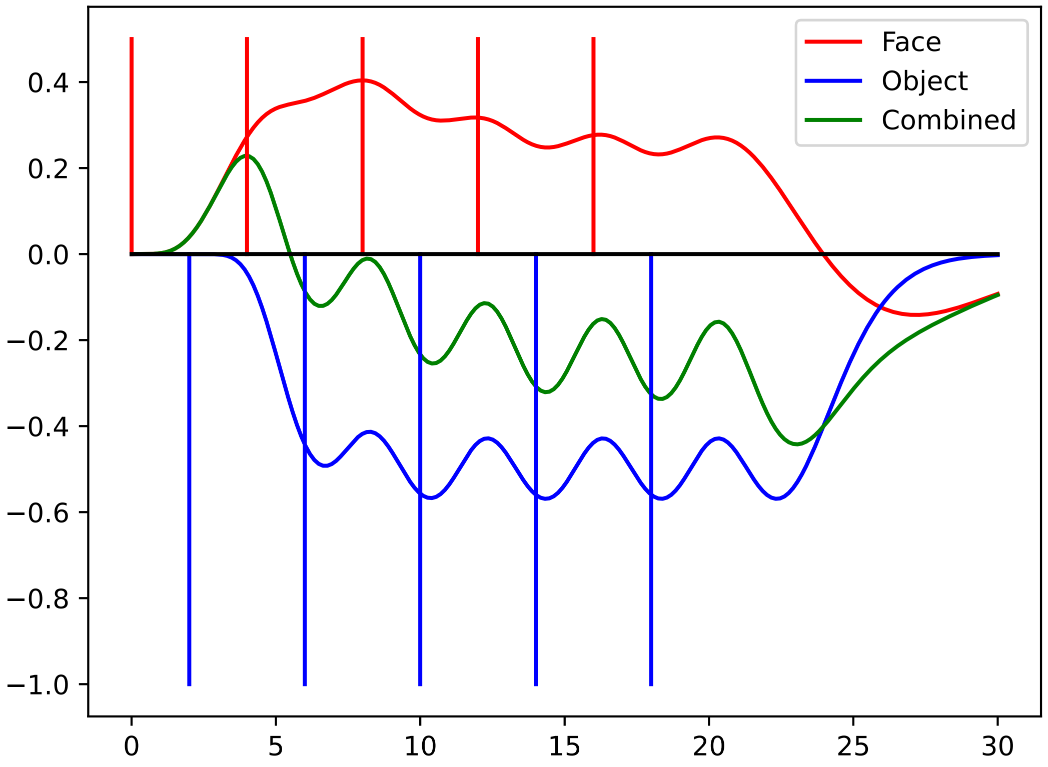
4.5.3 Derivative information

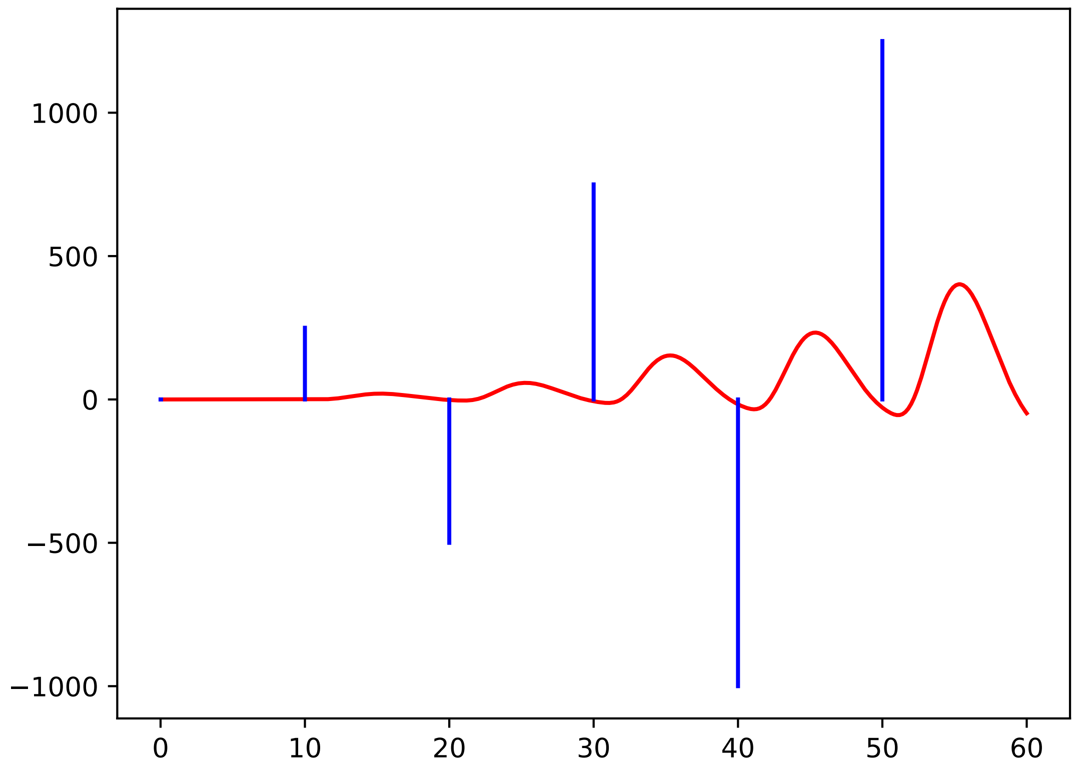
In cases where the neuronal model has more than one derivative, such as the continuous stimuli continuous-stimuli example, we might model the hemodynamic response using the higher derivatives as well. For example

$$h(t) = \beta_{O,0} \tilde{n}_{f_O}(t) + \beta_{O,1} \frac{\partial}{\partial t} \tilde{n}_{f_O}(t) + \beta_{C,0} \tilde{n}_{f_C}(t) + \beta_{C,1} \frac{\partial}{\partial t} \tilde{n}_{f_C}(t)$$

where

$$\begin{aligned} \tilde{n}_f(t) &= \frac{\partial}{\partial t} \tilde{N}_f((-\infty, t]) \\ &= \frac{\partial}{\partial t} \left(\int_{-\infty}^{t} \int_V f(v,t) \, dE(v,t) \right) \end{aligned}$$


Face
Object
Combined
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25
30



1000
500
0
−500
−1000
0
10
20
30
40
50
60

4.6 Design matrix

In a typical GLM analysis, we will compare the observed BOLD signal $B(t)$ at some fixed voxel x, observed at time points $(s_1, \dots, s_n)$, to a hemodynamic response model. For instance, in the *Face* vs. *Object* model, using the canonical HRF

$$B(t) = \beta_a \sum_{t_i : i \text{ odd}} h_{can}(t - t_i) + \beta_b \sum_{t_i : i \text{ even}} h_{can}(t - t_i) + \epsilon(t)$$

where $\epsilon(t)$ is the correlated noise in the BOLD data.

Because the BOLD is modeled as linear in (β_a, β_b) this fits into a multiple linear regression model setting, typically written as

$$Y_{n \times 1} = X_{n \times p} \beta_{p \times 1} + \epsilon_{n \times 1}$$

In order to fit the regression model, we must find the matrix X. This is just the derivative of the model of the mean of B with respect to the parameters to be estimated. Setting $(\beta_1, \beta_2) = (\beta_a, \beta_b)$

$$X_{ij} = \frac{\partial}{\partial \beta_j} \left(\beta_1 \sum_{t_k : k \text{ odd}} h_{can}(s_i - t_k) + \beta_b \sum_{t_k : k \text{ even}} h_{can}(s_i - t_k) \right)$$

4.6.1 Drift

We sometimes include a natural spline model of the drift here.

This changes the design matrix by adding more columns, one for each function in our model of the drift. In general, starting from some model of the mean the design matrix is the derivative of the model of the mean, differentiated with respect to all parameters to be estimated (in some fixed order).

4.6.2 Nonlinear example

The delayed continuous stimuli example above is an example of a nonlinear function of the mean that is nonlinear in some parameters, (τ_O, τ_C).

4.7 Formula objects

This experience of building the model can often be simplified, using what is known in :ref:R as *formula* objects. NiPy has implemented a formula object that is similar to R's, but differs in some important respects. See `nipy.algorithms.statistics.formula`.

CHAPTER FIVE

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 change. See 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 researchers 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 electro-physiological 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 software 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: Wikipedia GPL 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: Wikipedia MATLAB. It has good numerical algorithms, 2D graphics, and documentation. There are several large neuroscience software projects written 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 method of detecting the spatial distributions of certain radio-labeled 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 the Wikipedia voxel entry.

water activation PET
A *PET* technique to detect regional changes in blood flow. Before each scan, we inject the subject with radio-labeled water. The radio-labeled 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

NeuroSpin tools

The package `nipy.labs` hosts some tools that where originally developed at NeuroSpin, France. The list below also includes routines for estimating the empirical null, moved from `nipy.labs` to `nipy.algorithms.statistics`.

CHAPTER

SIX

MASK-EXTRACTION UTILITIES

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

compute_mask(mean_volume[, ...])	Compute a mask file from fMRI data in 3D or 4D ndarrays.
compute_mask_files(input_filename[, ...])	Compute a mask file from fMRI nifti file(s)
compute_mask_sessions(session_images[, m, ...])	Compute a common mask for several sessions of fMRI data.

6.1 nipy.labs.utils.mask.compute_mask

`nipy.labs.utils.mask.`**`compute_mask`**(*mean_volume*, *reference_volume=None*, *m=0.2*, *M=0.9*, *cc=True*, *opening=2*, *exclude_zeros=False*)

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.

opening: int, optional
if opening is larger than 0, an morphological opening is performed, to keep only large structures. This step is useful to remove parts of the skull that might have been included.

exclude_zeros: boolean, optional
Consider zeros as missing values for the computation of the threshold. This option is useful if the images have been resliced with a large padding of zeros.

Returns

mask
[3D boolean ndarray] The brain mask

6.2 nipy.labs.utils.mask.compute_mask_files

`nipy.labs.utils.mask.`**`compute_mask_files`**(*input_filename*, *output_filename=None*, *return_mean=False*, *m=0.2*, *M=0.9*, *cc=1*, *exclude_zeros=False*, *opening=2*)

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).

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.

exclude_zeros: boolean, optional
Consider zeros as missing values for the computation of the threshold. This option is useful if the images have been resliced with a large padding of zeros.

opening: int, optional
Size of the morphological opening performed as post-processing

Returns

mask
[3D boolean array] 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.

6.3 nipy.labs.utils.mask.compute_mask_sessions

nipy.labs.utils.mask.**compute_mask_sessions**(*session_images*, *m=0.2*, *M=0.9*, *cc=1*, *threshold=0.5*, *exclude_zeros=False*, *return_mean=False*, *opening=2*)

Compute a common mask for several sessions of fMRI data.

Uses the mask-finding algorithms 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_images
[list of (list of strings) or nipy image objects] A list of images/list of nifti filenames. Each inner list/image represents a session.

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.

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.

exclude_zeros: boolean, optional
Consider zeros as missing values for the computation of the threshold. This option is useful if the images have been resliced with a large padding of zeros.

return_mean: boolean, optional
if return_mean is True, the mean image across subjects is returned.

opening: int, optional,
size of the morphological opening

Returns

mask
[3D boolean ndarray] The brain mask

mean
[3D float array] The mean image

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

SEVEN

EMPIRICAL NULL

The *nipy.algorithms.statistics.empirical_pvalue* 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.

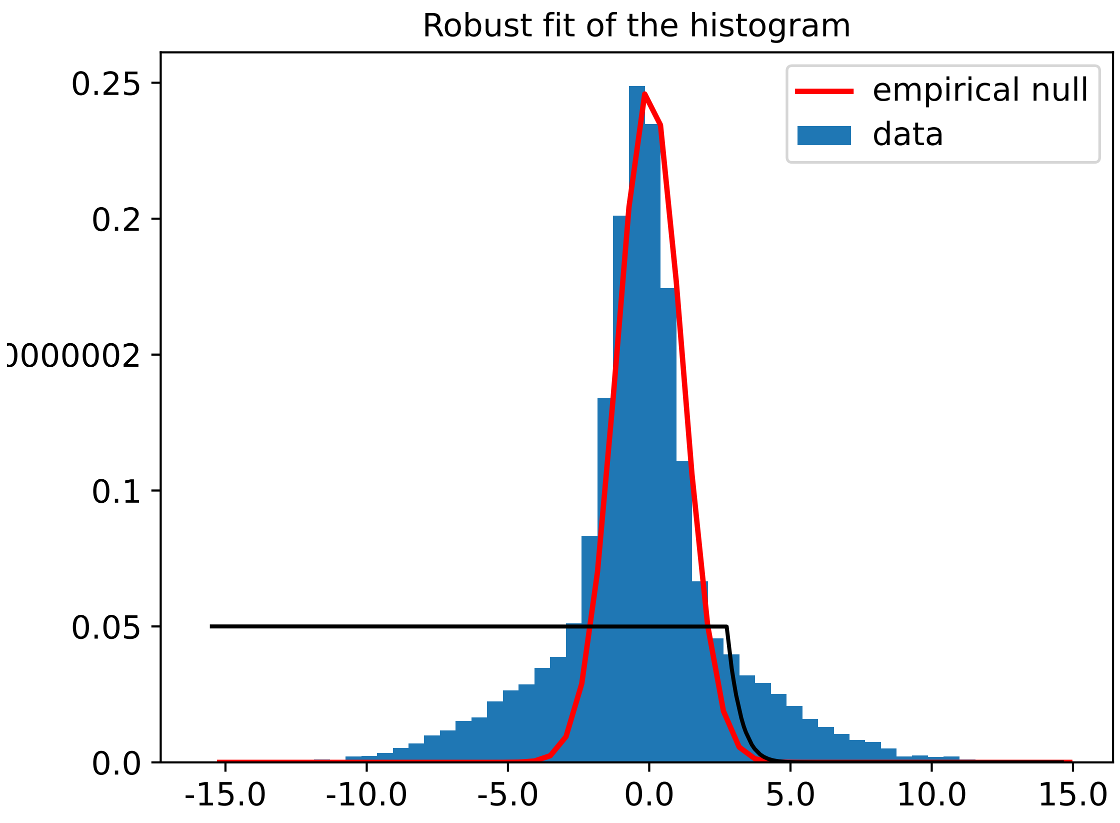
The *NormalEmpiricalNull* class learns its null distribution on the data provided at initialisation. Two different methods can be used to set a threshold from the null distribution: the *NormalEmpiricalNull.threshold()* method returns the threshold for a given false discovery rate, and thus accounts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

7.1 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 following which the data can be claimed to belong to the wide Gaussian:

```python
# emacs: -*- mode: python; py-indent-offset: 4; indent-tabs-mode: nil -*-
# vi: set ft=python sts=4 ts=4 sw=4 et:
import numpy as np

from nipy.algorithms.statistics.empirical_pvalue import NormalEmpiricalNull

x = np.c_[np.random.normal(size=10000),
          np.random.normal(scale=4, size=10000)]

enn = NormalEmpiricalNull(x)
enn.threshold(verbose=True)
```

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


Robust fit of the histogram
empirical null
data
0.25
0.2
)000002
0.1
0.05
0.0
-15.0
-10.0
-5.0
0.0
5.0
10.0
15.0

7.2 Class documentation

class `nipy.algorithms.statistics.empirical_pvalue.`**`NormalEmpiricalNull`**(*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*)

Initialize 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

Parameters

theta
[float or array of shape (n_samples)] values to test

Returns

afp
[value of array of shape(n)]

`fdrcurve`()

Returns the FDR associated with any point of self.x

`learn`(*left=0.2*, *right=0.8*)

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: variance of the estimated normal distribution
- sigma: np.sqrt(sqsigma) : standard deviation of the estimated normal distribution

`plot`(*efp=None*, *alpha=0.05*, *bar=1*, *mpaxes=None*)

Plot the histogram of x

Parameters

efp
[float, optional] The empirical FDR (corresponding to x) if efp==None, the false positive rate threshold plot is not drawn.

alpha
[float, optional] The chosen FDR threshold

bar=1
[bool, optional]

mpaxes=None: if not None, handle to an axes where the fig will be drawn. Avoids creating unnecessarily new figures

`threshold`(*alpha=0.05*, *verbose=0*)

Compute the threshold corresponding 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.

Returns

theta: float
the critical value associated with the provided FDR

`uncorrected_threshold`(*alpha=0.001*, *verbose=0*)

Compute the threshold corresponding to a specificity alpha for x

Parameters

alpha
[float, optional] the chosen false discovery rate (FDR) threshold.

verbose
[boolean, optional] the verbosity level, if True a plot is generated.

Returns

theta: float
the critical value associated with the provided p-value

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

CHAPTER

EIGHT

PLOTTING OF ACTIVATION MAPS

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

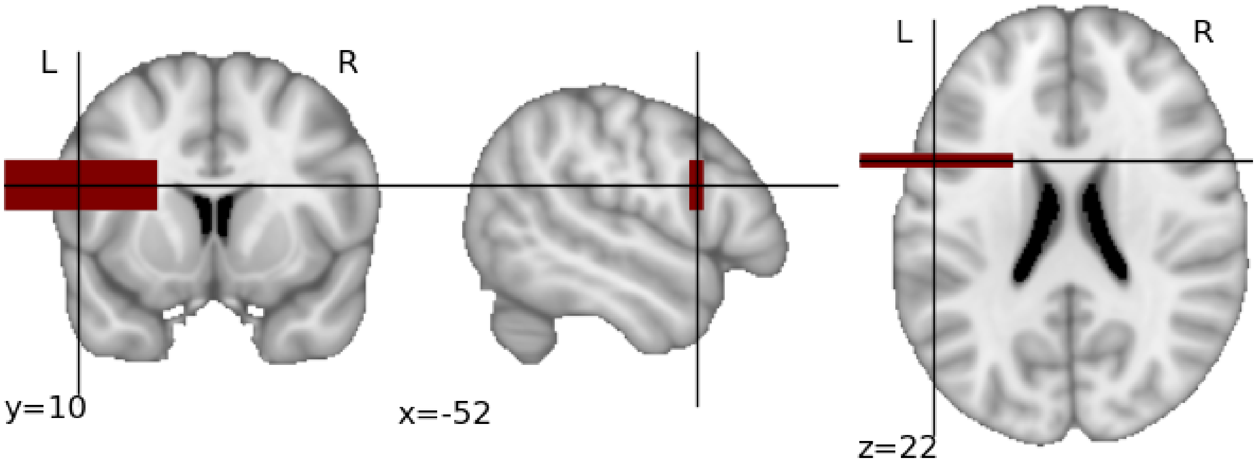
2D cuts of an activation map can be plotted and superimposed on an anatomical map using matplotlib. In addition, Mayavi2 can be used to plot 3D maps, using volumetric rendering. Some emphasis is made on automatic choice of default parameters, such as cut coordinates, to give a sensible view of a map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

Warning: The content of the module will change over time, as neuroimaging volumetric data structures are used instead of plain numpy arrays.

8.1 An example

```
from nipy.labs.viz 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 Broca Area
import numpy as np
mni_sform_inv = np.linalg.inv(mni_sform)
# Color an asymmetric rectangle around Broca area:
x, y, z = -52, 10, 22
x_map, y_map, z_map = coord_transform(x, y, z, mni_sform_inv)
map = np.zeros((182, 218, 182))
map[x_map-30:x_map+30, y_map-3:y_map+3, z_map-10:z_map+10] = 1

# We use a masked array to add transparency to the parts that we are
# not interested in:
thresholded_map = np.ma.masked_less(map, 0.5)

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

This creates the following image:



The same plot can be obtained fully automatically, by letting *plot_map()* find the activation threshold and the cut coordinates:

```
plot_map(map, mni_sform, threshold='auto')
```

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

8.2 *nipy.labs.viz* functions

plot_map(map, affine[, cut_coords, anat, ...])	Plot three cuts of a given activation map (Frontal, Axial, and Lateral)

8.2.1 nipy.labs.viz_tools.activation_maps.plot_map

nipy.labs.viz_tools.activation_maps.**plot_map**(*map*, *affine*, *cut_coords=None*, *anat=None*, *anat_affine=None*, *slicer='ortho'*, *figure=None*, *axes=None*, *title=None*, *threshold=None*, *annotate=True*, *draw_cross=True*, *do3d=False*, *threshold_3d=None*, *view_3d=(38.5, 70.5, 300, (-2.7, -12, 9.1))*, *black_bg=False*, ***imshow_kwargs*)

Plot three cuts of a given activation map (Frontal, Axial, and Lateral)

Parameters

map
[3D ndarray] The activation map, as a 3D image.

affine
[4x4 ndarray] The affine matrix going from image voxel space to MNI space.

cut_coords: None, int, or a tuple of floats
The MNI coordinates of the point where the cut is performed, in MNI coordinates and order. If slicer is ‘ortho’, this should be a 3-tuple: (x, y, z) For slicer == ‘x’, ‘y’, or ‘z’, then these are the coordinates of each cut in the corresponding direction. If None or an int is given, then a maximally separated sequence (with exactly cut_coords elements if cut_coords is not None) of cut coordinates along the slicer axis is computed automatically

anat
[3D ndarray or False, optional] The anatomical image to be used as a background. If None, the MNI152 T1 1mm template is used. If False, no anat is displayed.

anat_affine
[4x4 ndarray, optional] The affine matrix going from the anatomical image voxel space to MNI space. This parameter is not used when the default anatomical is used, but it is compulsory when using an explicit anatomical image.

slicer: {'ortho', 'x', 'y', 'z'}
Choose the direction of the cuts. With 'ortho' three cuts are performed in orthogonal directions

figure
[integer or matplotlib figure, optional] Matplotlib figure used or its number. If None is given, a new figure is created.

axes
[matplotlib axes or 4 tuple of float: (xmin, ymin, width, height), optional] The axes, or the coordinates, in matplotlib figure space, of the axes used to display the plot. If None, the complete figure is used.

title
[string, optional] The title displayed on the figure.

threshold
[a number, None, or 'auto'] If None is given, the maps are not thresholded. If a number is given, it is used to threshold the maps: values below the threshold are plotted as transparent. If auto is given, the threshold is determined magically by analysis of the map.

annotate: boolean, optional
If annotate is True, positions and left/right annotation are added to the plot.

draw_cross: boolean, optional
If draw_cross is True, a cross is drawn on the plot to indicate the cut plosition.

do3d: {True, False or 'interactive'}, optional
If True, Mayavi is used to plot a 3D view of the map in addition to the slicing. If 'interactive', the 3D visualization is displayed in an additional interactive window.

threshold_3d:
The threshold to use for the 3D view (if any). Defaults to the same threshold as that used for the 2D view.

view_3d: tuple,
The view used to take the screenshot: azimuth, elevation, distance and focalpoint, see the docstring of mlab.view.

black_bg: boolean, optional
If True, the background of the image is set to be black. If you wish to save figures with a black background, you will need to pass "facecolor='k', edgecolor='k'" to pyplot's savefig.

imshow_kwargs: extra keyword arguments, optional
Extra keyword arguments passed to pyplot.imshow

Notes

Arrays should be passed in numpy convention: (x, y, z) ordered.

Use masked arrays to create transparency:

import numpy as np map = np.ma.masked_less(map, 0.5) plot_map(map, affine)

8.3 3D plotting utilities

The module `nipy.labs.viz3d` can be used as helpers to represent neuroimaging volumes with Mayavi2.

plot_map_3d(map, affine[, cut_coords, anat, ...])	Plot a 3D volume rendering view of the activation, with an outline of the brain.
plot_anat_3d([anat, anat_affine, scale, ...])	3D anatomical display

8.3.1 nipy.labs.viz_tools.maps_3d.plot_map_3d

`nipy.labs.viz_tools.maps_3d.`**`plot_map_3d`**(*map*, *affine*, *cut_coords=None*, *anat=None*, *anat_affine=None*, *threshold=None*, *offscreen=False*, *vmin=None*, *vmax=None*, *cmap=None*, *view=(38.5, 70.5, 300, (-2.7, -12, 9.1))*)

Plot a 3D volume rendering view of the activation, with an outline of the brain.

Parameters

map
[3D ndarray] The activation map, as a 3D image.

affine
[4x4 ndarray] The affine matrix going from image voxel space to MNI space.

cut_coords: 3-tuple of floats, optional
The MNI coordinates of a 3D cursor to indicate a feature or a cut, in MNI coordinates and order.

anat
[3D ndarray, optional] The anatomical image to be used as a background. If None, the MNI152 T1 1mm template is used. If False, no anatomical image is used.

anat_affine
[4x4 ndarray, optional] The affine matrix going from the anatomical image voxel space to MNI space. This parameter is not used when the default anatomical is used, but it is compulsory when using an explicit anatomical image.

threshold
[float, optional] The lower threshold of the positive activation. This parameter is used to threshold the activation map.

offscreen: boolean, optional
If True, Mayavi attempts to plot offscreen. Will work only with VTK >= 5.2.

vmin
[float, optional] The minimal value, for the colormap

vmax
[float, optional] The maximum value, for the colormap

cmap
[a callable, or a pyplot colormap] A callable returning a (n, 4) array for n values between 0 and 1 for the colors. This can be for instance a pyplot colormap.

Notes

If you are using a VTK version below 5.2, there is no way to avoid opening a window during the rendering under Linux. This is necessary to use the graphics card for the rendering. You must maintain this window on top of others and on the screen.

8.3.2 nipy.labs.viz_tools.maps_3d.plot_anat_3d

`nipy.labs.viz_tools.maps_3d.`**`plot_anat_3d`**(*anat=None*, *anat_affine=None*, *scale=1*, *sulci_opacity=0.5*, *gyri_opacity=0.3*, *opacity=None*, *skull_percentile=78*, *wm_percentile=79*, *outline_color=None*)

3D anatomical display

Parameters

skull_percentile
[float, optional] The percentile of the values in the image that delimit the skull from the outside of the brain. The smaller the fraction of you field of view is occupied by the brain, the larger this value should be.

wm_percentile
[float, optional] The percentile of the values in the image that delimit the white matter from the grey matter. Typical this is skull_percentile + 1

For more versatile visualizations the core idea is that given a 3D map and an affine, the data is exposed in Mayavi as a volumetric source, with world space coordinates corresponding to figure coordinates. Visualization modules can be applied on this data source as explained in the Mayavi manual

`affine_img_src`(data, affine[, scale, name, ...])	Make a Mayavi source defined by a 3D array and an affine, for which the voxel of the 3D array are mapped by the affine.

8.3.3 nipy.labs.viz_tools.maps_3d.affine_img_src

`nipy.labs.viz_tools.maps_3d.`**`affine_img_src`**(*data*, *affine*, *scale=1*, *name='AffineImage'*, *reverse_x=False*)

Make a Mayavi source defined by a 3D array and an affine, for which the voxel of the 3D array are mapped by the affine.

Parameters

data: 3D ndarray
The data arrays

affine: (4 x 4) ndarray
The (4 x 4) affine matrix relating voxels to world coordinates.

scale: float, optional
An optional addition scaling factor.

name: string, optional
The name of the Mayavi source created.

reverse_x: boolean, optional
Reverse the x (lateral) axis. Useful to compared with images in radiologic convention.

Notes

The affine should be diagonal.

CHAPTER

NINE

GENERATING SIMULATED ACTIVATION MAPS

The module *nipy.labs.utils.simul_multisubject_fmri_dataset* contains a various functions to create simulated activation maps in two, three and four dimensions. A 2D example is *surrogate_2d_dataset()*. The functions can position various activations and add noise, both as background noise and jitter in the activation positions and amplitude.

These functions can be useful to test methods.

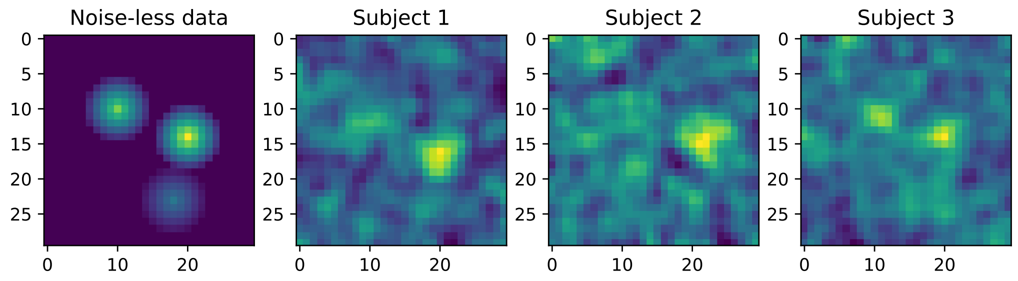
9.1 Example

```
# emacs: -*- mode: python; py-indent-offset: 4; indent-tabs-mode: nil -*-
# vi: set ft=python sts=4 ts=4 sw=4 et:
import numpy as np
import pylab as pl

from nipy.labs.utils.simul_multisubject_fmri_dataset import surrogate_2d_dataset

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

# First generate some noiseless data
noiseless_data = surrogate_2d_dataset(n_subj=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 = surrogate_2d_dataset(n_subj=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])
```

(continues on next page)

(continued from previous page)

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



9.2 Function documentation

`nipy.labs.utils.simul_multisubject_fmri_dataset.`**`surrogate_2d_dataset`**(*n_subj=10*, *shape=(30, 30)*, *sk=1.0*, *noise_level=1.0*, *pos=array([[6, 7], [10, 10], [15, 10]])*, *ampli=array([3, 4, 4])*, *spatial_jitter=1.0*, *signal_jitter=1.0*, *width=5.0*, *width_jitter=0*, *out_text_file=None*, *out_image_file=None*, *seed=False*)

Create surrogate (simulated) 2D activation data with spatial noise

Parameters

n_subj: integer, optional
The number of subjects, ie the number of different maps generated.

shape=(30,30): tuple of integers,
the shape of each image

sk: float, optional
Amount of spatial noise smoothness.

noise_level: float, optional
Amplitude of the spatial noise. amplitude=noise_level)

pos: 2D ndarray of integers, optional
x, y positions of the various simulated activations.

ampli: 1D ndarray of floats, optional
Respective amplitude of each activation

spatial_jitter: float, optional
Random spatial jitter added to the position of each activation, in pixel.

signal_jitter: float, optional
Random amplitude fluctuation for each activation, added to the amplitude specified by *ampli*

width: float or ndarray, optional
Width of the activations

width_jitter: float
Relative width jitter of the blobs

out_text_file: string or None, optional
If not None, the resulting array is saved as a text file with the given file name

out_image_file: string or None, optional
If not None, the resulting is saved as a nifti file with the given file name.

seed=False: int, optional
If seed is not False, the random number generator is initialized at a certain value

Returns

dataset: 3D ndarray
The surrogate activation map, with dimensions `(n_subj,) + shape`

`nipy.labs.utils.simul_multisubject_fmri_dataset.`**`surrogate_3d_dataset`**(*n_subj=1*, *shape=(20, 20, 20)*, *mask=None*, *sk=1.0*, *noise_level=1.0*, *pos=None*, *ampli=None*, *spatial_jitter=1.0*, *signal_jitter=1.0*, *width=5.0*, *out_text_file=None*, *out_image_file=None*, *seed=False*)

Create surrogate (simulated) 3D activation data with spatial noise.

Parameters

n_subj: integer, optional
The number of subjects, ie the number of different maps generated.

shape=(20,20,20): tuple of 3 integers,
the shape of each image

mask=None: Nifti1Image instance,
referential- and mask- defining image (overrides shape)

sk: float, optional
Amount of spatial noise smoothness.

noise_level: float, optional
Amplitude of the spatial noise. amplitude=noise_level)

pos: 2D ndarray of integers, optional
x, y positions of the various simulated activations.

ampli: 1D ndarray of floats, optional
Respective amplitude of each activation

spatial_jitter: float, optional
Random spatial jitter added to the position of each activation, in pixel.

signal_jitter: float, optional
Random amplitude fluctuation for each activation, added to the amplitude specified by ampli

width: float or ndarray, optional
Width of the activations

out_text_file: string or None, optional
If not None, the resulting array is saved as a text file with the given file name

out_image_file: string or None, optional
If not None, the resulting is saved as a nifti file with the given file name.

seed=False: int, optional
If seed is not False, the random number generator is initialized at a certain value

Returns

dataset: 3D ndarray
The surrogate activation map, with dimensions `(n_subj,) + shape`

`nipy.labs.utils.simul_multisubject_fmri_dataset.`**`surrogate_4d_dataset`**(*shape=(20, 20, 20)*, *mask=None*, *n_scans=1*, *n_sess=1*, *dmtx=None*, *sk=1.0*, *noise_level=1.0*, *signal_level=1.0*, *out_image_file=None*, *seed=False*)

Create surrogate (simulated) 3D activation data with spatial noise.

Parameters

shape = (20, 20, 20): tuple of integers,
the shape of each image

mask=None: brifti image instance,
referential- and mask- defining image (overrides shape)

n_scans: int, optional,
number of scans to be simlulated overridden by the design matrix

n_sess: int, optional,
the number of simulated sessions

dmtx: array of shape(n_scans, n_rows),
the design matrix

sk: float, optional
Amount of spatial noise smoothness.

noise_level: float, optional
Amplitude of the spatial noise. amplitude=noise_level)

signal_level: float, optional,
Amplitude of the signal

out_image_file: string or list of strings or None, optional
If not None, the resulting is saved as (set of) nifti file(s) with the given file path(s)

seed=False: int, optional
If seed is not False, the random number generator is initialized at a certain value

Returns

dataset: a list of n_sess ndarray of shape
(shape[0], shape[1], shape[2], n_scans) The surrogate activation map

CHAPTER

TEN

VOLUMETRIC DATA STRUCTURES

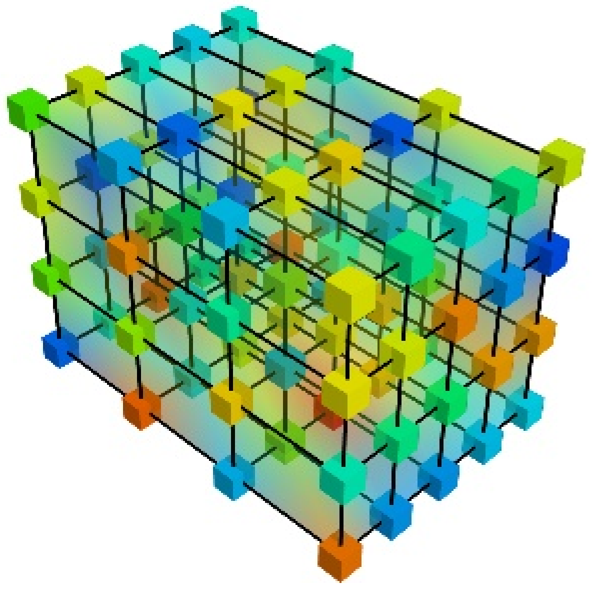
Volumetric data structures expose numerical values embedded in a world space. For instance, a volume could expose the T1 intensity, as acquired in scanner space, or the BOLD signal in MNI152 template space. The values can be multi-dimensional, in the case of a BOLD signal, the fMRI signal would correspond to a time series at each position in world space.

10.1 The image structure: `VolumeImg`

The structure most often used in neuroimaging is the *`VolumeImg`*. It corresponds, for instance, to the structure used in the Nifti files. This structure stores data as an n-dimensional array, with n being at least 3, alongside with the necessary information to map it to world space.

definition

A volume-image (class: *`VolumeImg`*) is a volumetric datastructure given by data points lying on a regular grid: this structure is a generalization of an image in 3D. The voxels, vertices of the grid, are mapped to coordinates by an affine transformation. As a result, the grid is regular and evenly-spaced, but may not be orthogonal, and the spacing may differ in the 3 directions.



The data is exposed in a multi dimensional array, with the 3 first axis corresponding to spatial directions. A complete description of this object can be found on the page: *`VolumeImg`*.

10.2 Useful methods on volume structures

Any general volume structures will implement methods for querying the values and changing world space (see the *VolumeField* documentation for more details):

VolumeField.values_in_world(x, y, z[, ...])	Return the values of the data at the world-space positions given by x, y, z
VolumeField.composed_with_transform(...)	Return a new image embedding the same data in a different word space using the given world to world transform.

10.2.1 nipy.labs.datasets.volumes.volume_field.VolumeField.values_in_world

VolumeField.**values_in_world**(*x*, *y*, *z*, *interpolation=None*)

Return the values of the data at the world-space positions given by x, y, z

Parameters

x
[number or ndarray] x positions in world space, in other words millimeters

y
[number or ndarray] y positions in world space, in other words millimeters. The shape of y should match the shape of x

z
[number or ndarray] z positions in world space, in other words millimeters. The shape of z should match the shape of x

interpolation
[None, ‘continuous’ or ‘nearest’, optional] Interpolation type used when calculating values in different word spaces. If None, the image’s interpolation logic is used.

Returns

values
[number or ndarray] Data values interpolated at the given world position. This is a number or an ndarray, depending on the shape of the input coordinate.

10.2.2 nipy.labs.datasets.volumes.volume_field.VolumeField.composed_with_transform

VolumeField.**composed_with_transform**(*w2w_transform*)

Return a new image embedding the same data in a different word space using the given world to world transform.

Parameters

w2w_transform
[transform object] The transform object giving the mapping between the current world space of the image, and the new word space.

Returns

remapped_image
[nipy image] An image containing the same data, expressed in the new world space.

Also, as volumes structure may describe the spatial data in various way, you can easily to convert to a `VolumeImg`, ie a regular grid, for instance to do implement an algorithm on the grid such as spatial smoothing:

`VolumeField.as_volume_img`([affine, shape, ...])	Resample the image to be an image with the data points lying on a regular grid with an affine mapping to the word space (a nipy VolumeImg).

10.2.3 nipy.labs.datasets.volumes.volume_field.VolumeField.as_volume_img

`VolumeField.`**`as_volume_img`**(*affine=None*, *shape=None*, *interpolation=None*, *copy=True*)

Resample the image to be an image with the data points lying on a regular grid with an affine mapping to the word space (a nipy VolumeImg).

Parameters

affine: 4x4 or 3x3 ndarray, optional
Affine of the new voxel grid or transform object pointing to the new voxel coordinate grid. If a 3x3 ndarray is given, it is considered to be the rotation part of the affine, and the best possible bounding box is calculated, in this case, the shape argument is not used. If None is given, a default affine is provided by the image.

shape: (n_x, n_y, n_z), tuple of integers, optional
The shape of the grid used for sampling, if None is given, a default affine is provided by the image.

interpolation
[None, ‘continuous’ or ‘nearest’, optional] Interpolation type used when calculating values in different word spaces. If None, the image’s interpolation logic is used.

Returns

resampled_image
[nipy VolumeImg] New nipy VolumeImg with the data sampled on the grid defined by the affine and shape.

Notes

The coordinate system of the image is not changed: the returned image points to the same world space.

Finally, different structures can embed the data differently in the same world space, for instance with different resolution. You can resample one structure on another using:

`VolumeField.resampled_to_img`(target_image[, ...])	Resample the volume to be sampled similarly than the target volumetric structure.

10.2.4 nipy.labs.datasets.volumes.volume_field.VolumeField.resampled_to_img

`VolumeField.`**`resampled_to_img`**`(`*target_image*, *interpolation=None*`)`

Resample the volume to be sampled similarly than the target volumetric structure.

Parameters

target_image
[nipy volume] Nipy volume structure onto the grid of which the data will be resampled.

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the volume's interpolation logic is used.

Returns

resampled_image
[nipy_image] New nipy image with the data resampled.

Notes

Both the target image and the original image should be embedded in the same world space.

FIXME: Examples would be good here, but first we need io and template data to be wired with datasets.

10.3 More general data structures

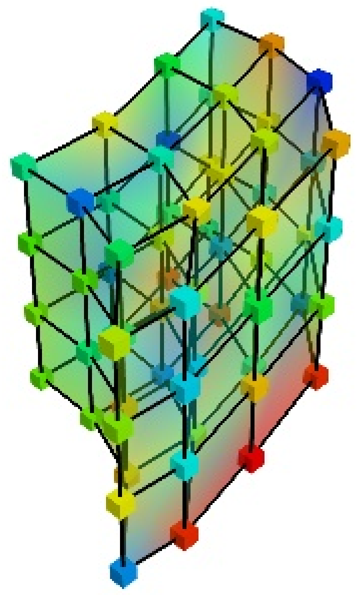
The *`VolumeImg`* is the most commonly found volume structure, and the simplest to understand, however, volumetric data can be described in more generic terms, and for performance reason it might be interesting to use other objects.

Here, we give a list of the nipy volumetric data structures, from most specific, to most general. When you deal with volume structures in your algorithms, depending on which volume structure class you are taking as an input, you can assume different properties of the data. You can always use *`VolumeImg.as_volume_img()`* to cast the volume structure in a *`VolumeImg`* that is simple to understand and easy to work with, but it may not be necessary.

10.3.1 Implemented classes

Implemented classes (or *concrete* classes) are structures that you can readily use directly from nipy.

`VolumeGrid`
In a *`VolumeGrid`*, the data points are sampled on a 3D grid, but unlike for a `VolumeImg`, grid may not be regular. For instance, it can be a grid that has been warped by a non-affine transformation. Like with the `VolumeImg`, the data is exposed in a multi dimensional array, with the 3 first axis corresponding to spatial directions.

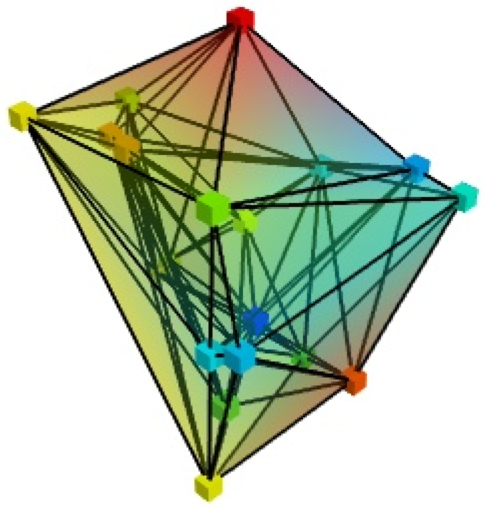


10.3.2 Abstract classes

Abstract classes cannot be used because they are incompletely implemented. They serve as to define the interface: the type of objects that you can use, or how you can extend nipy by exposing the same set of methods and attributes (the *interface*).

VolumeData

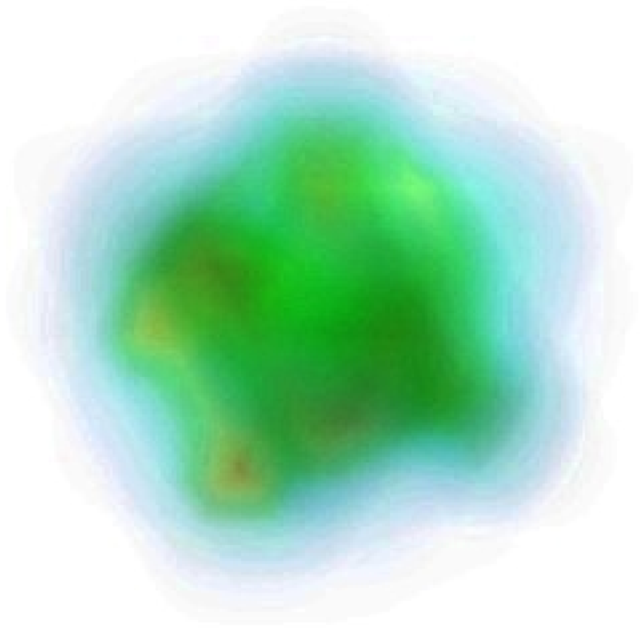
In this volumetric structure, the data is sampled for some points in the world space. The object knows how to interpolate between these points. The underlying values are stored in a multidimensional array-like object that can be indexed and sliced.



This is an abstract base class: it defines an interface, but is not fully functional, and can be used only via its children class (such as `VolumeGrid` or `VolumeImg`).

VolumeField

This is the most general volumetric structure (base class): all the nipy volume expose this interface. This structure does not make any assumptions on how the values are internal represented, they may, for instance, be represented as a function, rather than as data points, or as a data structure that is not an array, such as a graph.



This is also an abstract base class: it defines the core nipy volumetric data structure interface: you can rely on all the methods documented for this class in any nipy data structure.

Part III

Developer Guide

CHAPTER

ELEVEN

DEVELOPMENT QUICKSTART

11.1 Source Code

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

11.2 Checking out the latest version

To check out the latest version of nipy you need git:

```
git clone git://github.com/nipy/nipy.git
```

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. Create 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.

11.3 Getting data files

See data_files.

11.4 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.
- *Development 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."

11.5 Submitting a patch

The preferred method to submit a patch is to create a branch of nipy on your machine, modify the code and make a patch or patches. Then email the nipy mailing list and we will review your code and hopefully apply (merge) your patch. See the instructions for *Making patches*.

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

11.6 Bug reports

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

CHAPTER

TWELVE

DEVELOPER INSTALLS FOR DIFFERENT DISTRIBUTIONS

12.1 Debian / Ubuntu developer install

12.1.1 Dependencies

See *Download and Install* for the installation instructions. Since NiPy is provided within stock distribution (`main` component of Debian, and `universe` of Ubuntu), to install all necessary requirements it is enough to:

```
sudo apt-get build-dep python-nipy
```

Note: Above invocation assumes that you have references to `Source` repository listed with `deb-src` prefixes in your apt .list files.

Otherwise, you can revert to manual installation with:

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

12.1.2 Useful additions

Some functionality in NiPy requires additional modules:

```
sudo apt-get install ipython
sudo apt-get install python-matplotlib
sudo apt-get install mayavi2
```

For getting the code via version control:

```
sudo apt-get install git-core
```

Then follow the instructions at *Submitting a patch*.

And for easier control of multiple Python modules installations (e.g. different versions of IPython):

```
sudo apt-get install virtualenvwrapper
```

12.2 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
yum install sympy
yum install atlas-devel
```

Options:

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

For getting the code via version control:

```
yum install git-core
```

Then follow the instructions at *Submitting a patch*

12.3 Development install on windows

12.3.1 The easy way - a super-package

The easiest way to get the dependencies is to install PythonXY or the Enthought Tool Suite . This gives you MinGW, Python, Numpy, Scipy, ipython and matplotlib (and much more).

12.3.2 The hard way - by components

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
- Download and install Sympy

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 windows_scipy_build for details.

12.3.3 Getting and installing NIPY

You will next need to get the NIPY code via version control:

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

You should now be able to follow the instructions in *Submitting a patch*, but with the following modifications:

12.3.4 Running the build / install

Here we assume that you do *not* have the Microsoft visual C tools, you did not use the ETS package (which sets the compiler for you) and *are* using a version of MinGW to compile NIPY.

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

```
python setup.py build --compiler=mingw32 install
```

Note that, with this setup you cannot do inplace (developer) installs (like `python setup.py build_ext --inplace`) because of a six-legged python packaging feature that does not allow the compiler options (here `--compiler=mingw32`) to be passed from the `build_ext` command.

If you want to be able to do that, add these lines to your `distutils.cfg` file

```
[build]
compiler=mingw32

[config]
compiler = mingw32
```

See http://docs.python.org/install/#inst-config-files for details on this file. After you've done this, you can run the standard `python setup.py build_ext --inplace` command.

The command line from Windows

The default windows XP command line `cmd` is very basic. You might consider using the Cygwin bash shell, or 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 --compiler=mingw32
```

CHAPTER

THIRTEEN

DEVELOPMENT GUIDELINES

13.1 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.

13.2 Getting build dependencies

13.2.1 Building the documentation

You need to have Sphinx (version 0.6.2 or above) and graphviz (version 2.20 or greater).

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
```

13.2.2 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.

13.2.3 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*

13.2.4 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.

13.2.5 Documentation Problems

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

13.3 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*)

13.3.1 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

13.3.2 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

13.3.3 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 *reference 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 facilities. For example, see the module `matplotlib.backend_bases` documentation, or the class `LocationEvent`, or the method `mpl_connect()`.

13.3.4 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.

13.3.5 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:

```
# emacs: -*- mode: python; py-indent-offset: 4; indent-tabs-mode: nil -*-
# vi: set ft=python sts=4 ts=4 sw=4 et:
import matplotlib.pyplot as plt

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

13.3.6 Using math

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

$$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \frac{1}{8 \pi 2} \int^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 \left[\frac{ U^{2\beta}_{\delta_1 \rho_1} - \alpha^\prime_2U^{1\beta}_{\rho_1 \sigma_2} }{U^{0\beta}_{\rho_1 \sigma_2}}\right]$$

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.

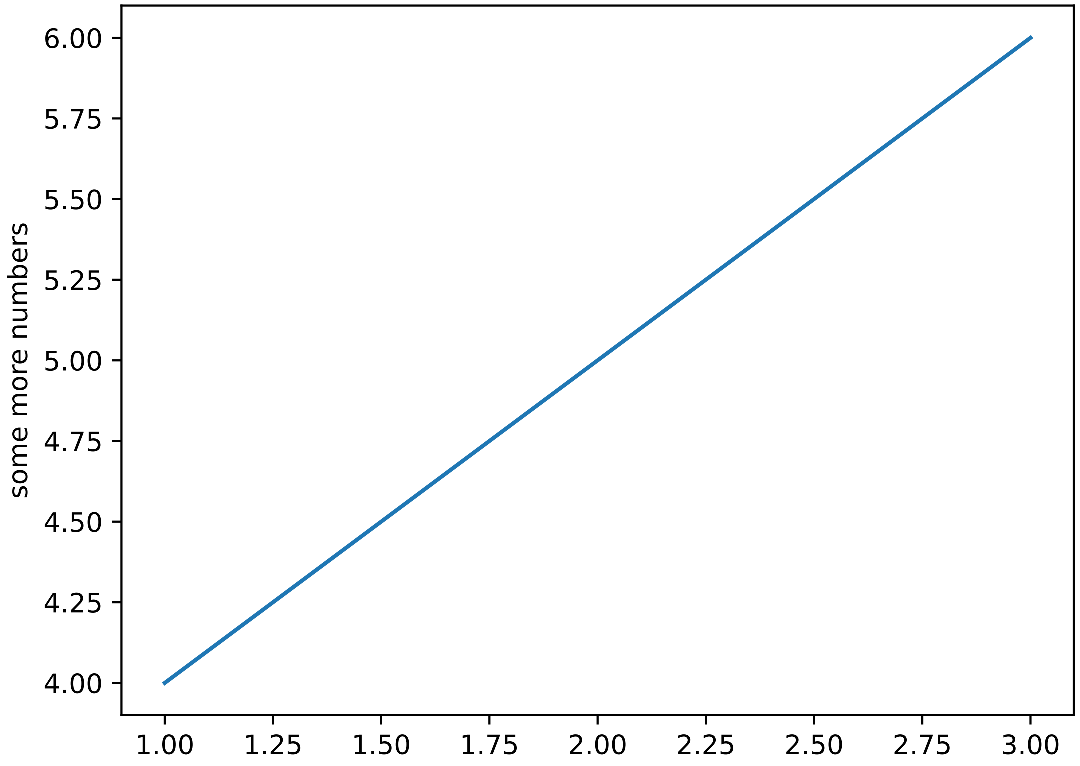
13.3.7 Inserting matplotlib plots

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.

```
# emacs: -*- mode: python; py-indent-offset: 4; indent-tabs-mode: nil -*-
# vi: set ft=python sts=4 ts=4 sw=4 et:
import matplotlib.pyplot as plt

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


some more numbers
6.00
5.75
5.50
5.25
5.00
4.75
4.50
4.25
4.00
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00

13.3.8 Emacs helpers

See *ReST mode*

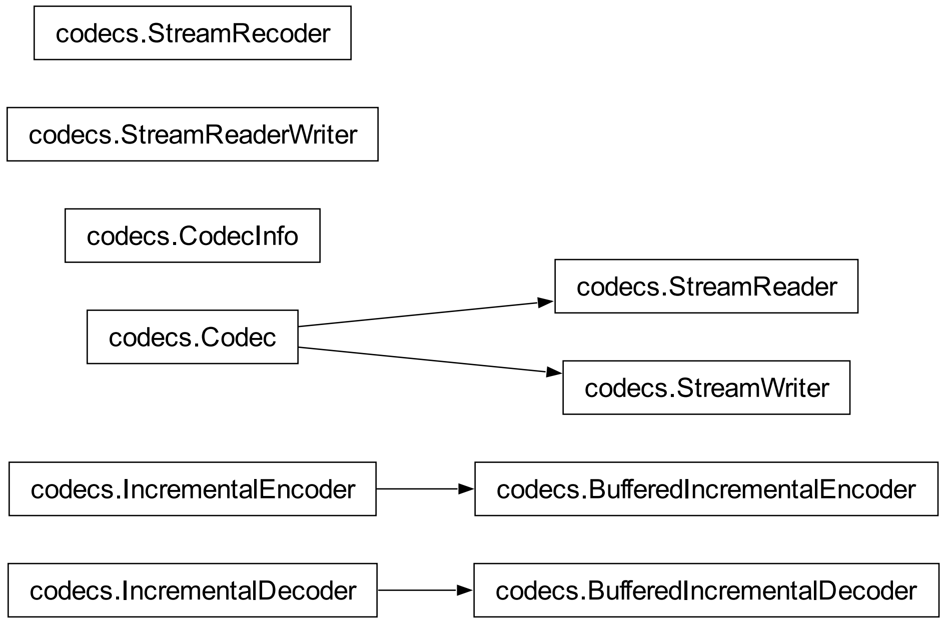
13.3.9 Inheritance diagrams

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:


codecs.StreamRecoder
codecs.StreamReaderWriter
codecs.CodecInfo
codecs.Codec
codecs.StreamReader
codecs.StreamWriter
codecs.IncrementalEncoder
codecs.BufferedIncrementalEncoder
codecs.IncrementalDecoder
codecs.BufferedIncrementalDecoder

13.3.10 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`)
```

(continues on next page)

(continued from previous page)

```


.. _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
```

(continues on next page)

(continued from previous page)

```
------------

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 *reference 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
```

(continues on next page)

(continued from previous page)

```
<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>`_ facilities.  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
```

(continues on next page)

(continued from previous page)

```
---------------

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^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 \left[\frac{ U^{2\beta}_{\delta_1 \rho_
↪1} - \alpha^\prime_2U^{1\beta}_{\rho_1 \sigma_2} }{U^{0\beta}_{\rho_1 \sigma_2}}\right]

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
```

(continues on next page)

(continued from previous page)

```
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
```

13.4 Working with *nipy* source code

Contents:

13.4.1 Introduction

These pages describe a git and github workflow for the **`nipy`_** project.

There are several different workflows here, for different ways of working with *nipy*.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see *git resources*.

13.4.2 Install git

Overview

Debian / Ubuntu	`sudo apt-get install git`
Fedora	`sudo yum install git`
Windows	Download and install msysGit
OS X	Use the git-osx-installer

In detail

See the git page for the most recent information.

Have a look at the github install help pages available from github help

There are good instructions here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

13.4.3 Following the latest source

These are the instructions if you just want to follow the latest *nipy* source, but you don't need to do any development for now.

The steps are:

- *Install git*
- get local copy of the nipy github git repository
- update local copy from time to time

Get the local copy of the code

From the command line:

```
git clone git://github.com/nipy/nipy.git
```

You now have a copy of the code tree in the new `nipy` directory.

Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd nipy
git pull
```

The tree in `nipy` will now have the latest changes from the initial repository.

13.4.4 Making a patch

You've discovered a bug or something else you want to change in **`nipy`_** .. — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the *Git for development* model instead.

Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/nipy/nipy.git
# make a branch for your patching
cd nipy
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C main
```

Then, send the generated patch files to the nipy mailing list — where we will thank you warmly.

In detail

1. Tell git who you are so it can label the commits you've made:

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

2. If you don't already have one, clone a copy of the **`nipy`_** repository:

   ```
   git clone git://github.com/nipy/nipy.git
   cd nipy
   ```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

   ```
   git branch the-fix-im-thinking-of
   git checkout the-fix-im-thinking-of
   ```

4. Do some edits, and commit them as you go:

   ```
   # hack, hack, hack
   # Tell git about any new files you've made
   git add somewhere/tests/test_my_bug.py
   # commit work in progress as you go
   git commit -am 'BF - added tests for Funny bug'
   # hack hack, hack
   git commit -am 'BF - added fix for Funny bug'
   ```

   Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag?.

5. When you have finished, check you have committed all your changes:

   ```
   git status
   ```

6. Finally, make your commits into patches. You want all the commits since you branched from the `main` branch:

   ```
   git format-patch -M -C main
   ```

   You will now have several files named for the commits:

   ```
   0001-BF-added-tests-for-Funny-bug.patch
   0002-BF-added-fix-for-Funny-bug.patch
   ```

   Send these files to the nipy mailing list.

When you are done, to switch back to the main copy of the code, just return to the `main` branch:

```
git checkout main
```

Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the **`nipy`_** repository on github — *Making your own copy (fork) of nipy*. Then:

```
# checkout and refresh main branch from main repo
git checkout main
git pull origin main
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/nipy.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

13.4.5 Git for development

Contents:

Making your own copy (fork) of nipy

You need to do this only once. The instructions here are very similar to the instructions at https://help.github.com/forking/ — please see that page for more detail. We're repeating some of it here just to give the specifics for the **`nipy`_** project, and to suggest some default names.

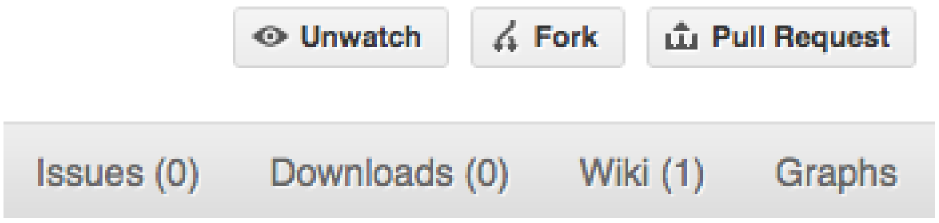
Set up and configure a github account

If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the `Generating SSH keys` help on github help.

Create your own forked copy of `nipy`_

1. Log into your github account.
2. Go to the **`nipy`_** github home at nipy github.
3. Click on the *fork* button:



Now, after a short pause, you should find yourself at the home page for your own forked copy of **`nipy`_**.

Set up your fork

First you follow the instructions for *Making your own copy (fork) of nipy*.

Overview

```
git clone git@github.com:your-user-name/nipy.git
cd nipy
git remote add upstream git://github.com/nipy/nipy.git
```

In detail

Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/nipy.git`
2. Investigate. Change directory to your new repo: `cd nipy`. Then `git branch -a` to show you all branches. You'll get something like:

   ```
   * main
   remotes/origin/main
   ```

   This tells you that you are currently on the `main` branch, and that you also have a `remote` connection to `origin/main`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

   Now you want to connect to the upstream nipy github repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

```
cd nipy
git remote add upstream git://github.com/nipy/nipy.git
```

`upstream` here is just the arbitrary name we're using to refer to the main **`nipy`_** repository at nipy github.

Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream     git://github.com/nipy/nipy.git (fetch)
upstream     git://github.com/nipy/nipy.git (push)
origin       git@github.com:your-user-name/nipy.git (fetch)
origin       git@github.com:your-user-name/nipy.git (push)
```

Configure git

Overview

Your personal git configurations are saved in the .gitconfig file in your home directory.

Here is an example .gitconfig file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com

[alias]
        ci = commit -a
        co = checkout
        st = status
        stat = status
        br = branch
        wdiff = diff --color-words

[core]
        editor = vim

[merge]
        summary = true
```

You can edit this file directly or you can use the git config --global command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your ~/.gitconfig file, or run the commands above.

In detail

user.name and user.email

It is good practice to tell git who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

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

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
      name = Your Name
      email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words
```

Editor

You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging

To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

Fancy log output

This is a very nice alias to get a fancy log output; it should go in the `alias` section of your `.gitconfig` file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
↪%C(bold blue)[%an]%Creset' --abbrev-commit --date=relative
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45␣
↪minutes ago) [Matthew Brett]
*   d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/
↪master (2 weeks ago) [Jonathan Terhorst]
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2 weeks␣
↪ago) [Corran Webster]
* 68f6752 - Initial implementation of AxisIndexer - uses 'index_by' which needs to be␣
↪changed to a call on an Axes object - this is all very sketchy right now. (2 weeks␣
↪ago) [Corr
*   376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan␣
↪Terhorst]
|\
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-axis␣
↪object (3 weeks ago) [Jonathan Terhorst]
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan Terhorst]
| *   956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago) [Jonathan␣
↪Terhorst]
| |\
| |/
```

Thanks to Yury V. Zaytsev for posting it.

Development workflow

You already have your own forked copy of the **`nipy`_** repository, by following *Making your own copy (fork) of nipy*. You have *Set up your fork*. You have configured git by following *Configure git*. Now you are ready for some real work.

Workflow summary

In what follows we'll refer to the upstream nipy `main` branch, as "trunk".

- Don't use your `main` branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — "one task, one branch" (ipython git workflow).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider *Rebasing on trunk*
- Ask on the nipy mailing list if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

See linux git workflow and ipython git workflow for some explanation.

Consider deleting your main branch

It may sound strange, but deleting your own `main` branch can help reduce confusion about which branch you are on. See **`deleting main on github`_** for details.

Update the mirror of trunk

First make sure you have done *Linking your repository to the upstream repo*.

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, 'trunk' is the branch referred to by (remote/branchname) `upstream/main` - and if there have been commits since you last checked, `upstream/main` will change after you do the fetch.

Make a new feature branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example add-ability-to-fly, or buxfix-for-issue-42.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/main
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public github fork of **`nipy`_**. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called origin. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the --set-upstream option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that my-new-feature is related to the my-new-feature branch in the github repo.

The editing workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with git status (see git status). You'll see a listing like this one:

```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   README
#
# Untracked files:
```

(continues on next page)

(continued from previous page)

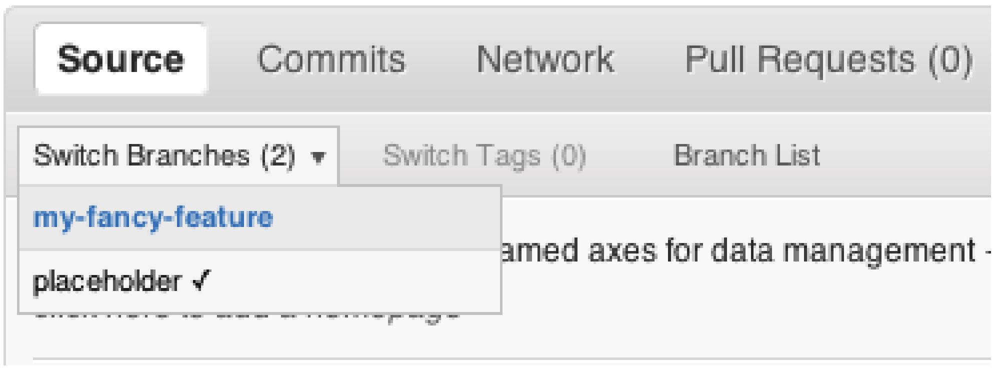
```
#   (use "git add <file>..." to include in what will be committed)
#
#  INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag? — and the helpful use-case description in the tangled working copy problem. The git commit manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see git push).

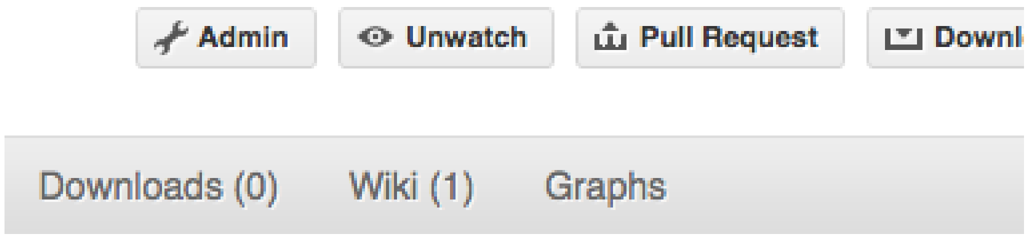
Ask for your changes to be reviewed or merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `https://github.com/your-user-name/nipy`.
2. Use the 'Switch Branches' dropdown menu near the top left of the page to select the branch with your changes:



3. Click on the 'Pull request' button:



Enter a title for the set of changes, and some explanation of what you've done. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with.

If you don't think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

Some other things you might want to do

Delete a branch on github

```
git checkout main
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

Note the colon `:` before `my-unwanted-branch`. See also: https://help.github.com/articles/pushing-to-a-remote/#deleting-a-remote-branch-or-tag

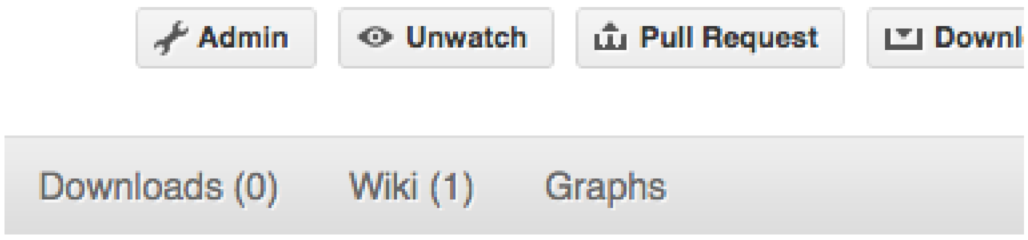
Several people sharing a single repository

If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork nipy into your account, as from *Making your own copy (fork) of nipy*.

Then, go to your forked repository github page, say `https://github.com/your-user-name/nipy`

Click on the 'Admin' button, and add anyone else to the repo as a collaborator:



Now all those people can do:

```
git clone git@githhub.com:your-user-name/nipy.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin main # pushes directly into your repo
```

Explore your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the network graph visualizer for your github repo.

Finally the *Fancy log output* `lg` alias will give you a reasonable text-based graph of the repository.

Rebasing on trunk

Let's say you thought of some work you'd like to do. You *Update the mirror of trunk* and *Make a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
     /
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
              A'--B'--C' cool-feature
             /
D---E---F---G trunk
```

See rebase without tears for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/main upstream/main cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/main
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recovering from mess-ups*.

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the "Description" section. There is some related help on merging in the git user manual - see resolving a merge.

Recovering from mess-ups

Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto␣
↪11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...

# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

Note: Do this only for your own feature branches.

There's an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and `6ad92e5` is the last commit in the `cool-feature` branch. Suppose we want to make the following changes:

- Rewrite the commit message for `13d7934` to something more sensible.
- Combine the commits `2dec1ac`, `a815645`, `eadc391` into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for `13d7934`, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained *above*.

Maintainer workflow

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository.

Being as how you're a maintainer, you are completely on top of the basic stuff in *Development workflow*.

The instructions in *Linking your repository to the upstream repo* add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:nipy/nipy.git
git fetch upstream-rw
```

Integrating changes

Let's say you have some changes that need to go into trunk (`upstream-rw/main`).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/nipy.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

A few commits

If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw
# rebase
git rebase upstream-rw/main
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

A long series of commits

If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/main
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history

Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/main..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/main`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to trunk

```
git push upstream-rw my-new-feature:main
```

This pushes the `my-new-feature` branch in this repository to the `main` branch in the `upstream-rw` repository.

13.4.6 git resources

Tutorials and summaries

- github help has an excellent series of how-to guides.
- The pro git book is a good in-depth book on git.
- A git cheat sheet is a page giving summaries of common commands.
- The git user manual
- The git tutorial
- The git community book
- git ready — a nice series of tutorials
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- git foundation expands on the git parable.
- Fernando Perez' git page — Fernando's git page — many links and tips

- A good but technical page on git concepts
- git svn crash course: git for those of us used to subversion

Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management
- Linus Torvalds on linux git workflow . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add
- git branch
- git checkout
- git clone
- git commit
- git config
- git diff
- git log
- git pull
- git push
- git remote
- git status

13.5 Commit message codes

Please prefix all commit summaries with one (or more) of the following labels. This should help others to easily classify the commits into meaningful categories:

- *BF* : bug fix
- *RF* : refactoring
- *ENH* : new feature or extended functionality
- *BW* : addresses backward-compatibility
- *OPT* : optimization
- *BK* : breaks something and/or tests fail
- *DOC*: for all kinds of documentation related commits

- *TEST* : for adding or changing tests
- *STY* : PEP8 conformance, whitespace changes etc that do not affect function.
- *WIP* : Work in progress; please try and avoid using this one, and rebase incomplete changes into functional units using e.g. `git rebase -i`

So your commit message might look something like this:

```
TEST: relax test threshold slightly

Attempted fix for failure on windows test run when arrays are in fact
very close (within 6 dp).
```

Keeping up a habit of doing this is useful because it makes it much easier to see at a glance which changes are likely to be important when you are looking for sources of bugs, fixes, large refactorings or new features.

13.6 Pull request codes

When you submit a pull request to github, github will ask you for a summary. If your code is not ready to merge, but you want to get feedback, please consider using `WIP - me working on image design` or similar for the title of your pull request. That way we will all know that it's not yet ready to merge and that you may be interested in more fundamental comments about design.

When you think the pull request is ready to merge, change the title (using the *Edit* button) to something like `MRG - my work on image design`.

13.7 Testing

Nipy uses the the Pytest framework. If you plan to do development on nipy please have a look at the Pytest docs and read through the numpy testing guidelines.

13.7.1 Automated testing

We run the tests on every commit with travis-ci |--| see nipy on travis.

We also have a farm of machines set up to run the tests on every commit to the `main` branch at nipy buildbot.

13.7.2 Writing tests

Test files

We like test modules to import their testing functions and classes from the module in which they are defined. For example, we might want to use the `assert_array_equal`, `assert_almost_equal` functions defined by `numpy`, and the `funcfile, anatfile` variables from `nipy`:

```
from numpy.testing import assert_array_equal, assert_almost_equal
from nipy.testing import funcfile, anatfile
```

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:

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

All tests go in a `tests` subdirectory for each package.

Temporary files

If you need to create a temporary file during your testing, you could use one of these three methods, in order of convenience:

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. *in_tmp_path* Pytest fixture.

   This is a convenient way of putting you into a temporary directory so you can save anything you like into the current directory, and feel fine about it after. Like this:

   ```
   def test_func(in_tmp_path):
       f = open('myfile', 'wt')
       f.write('Anything at all')
       f.close()
   ```

   One thing to be careful of is that you may need to delete objects holding onto the file before you exit the enclosing function, otherwise Windows may refuse to delete the file.

3. 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 on *Windows*.

   Example:

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

Please don't just create a file in the test directory and then remove it with a call to `os.remove`. For various reasons, sometimes `os.remove` doesn't get called and temp files get left around.

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 the method/function. For convenience, we often write many tests in a single test function. This has the disadvantage that if one test fails, the testing framework will not run any of the subsequent tests in the same function. This isn't a big problem in practice, because we run the tests so often (*Automated testing*) that we can quickly pick up and fix the failures.

For example, this test function executes four tests:

```
def test_index():
    cs = CoordinateSystem('ijk')
    assert_equal(cs.index('i'), 0)
    assert_equal(cs.index('j'), 1)
    assert_equal(cs.index('k'), 2)
    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. See the pytest documentation for various ways to do that, or search our code for *pytest.mark* for examples.

13.7.3 Running tests

Running the full test suite

To run nipy's tests, you will need to pytest installed. Then:

```
pytest nipy
```

You can run the full tests, including doctests with:

```
pip install pytest-doctestplus

pytest --doctest-plus nipy
```

Install optional data packages for testing

For our tests, we have collected a set of fmri imaging data which are required for the tests to run. To do this, download the latest example data and template package files from NIPY data packages. See data-files.

Running individual tests

You can also run the tests from the command line with a variety of options.

To test an individual module:

```
pytest nipy/core/image/tests/test_image.py
```

To test an individual function:

```
pytest nipy/core/image/tests/test_image.py::test_maxmin_values
```

To test a class:

```
pytest nipy/algorithms/clustering/tests/test_clustering.py::TestClustering
```

To test a class method:

```
pytest nipy/algorithms/clustering/tests/test_clustering.py::TestClustering.testkmeans1
```

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

```
pytest -sv nipy
```

To include doctests in the tests:

```
pytest -sv --docest-plus nipy
```

13.7.4 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 pytest - the automated test runner we are using. Pytest can run the python coverage tester. First make sure you have the coverage test plugin installed on your system:

```
pip install pytest-cov
```

Run Pytest with coverage testing arguments:

```
pytest --cov=nipy --doctest-plus nipy
```

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 tests. For instance, the default report also includes coverage of most of `numpy`. To focus the coverage report, you can provide Pytest with the specific package you would like output from using the `--cov=nipy` (the option above).

13.8 Debugging

Some options are:

13.8.1 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.

13.8.2 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 introspection, add:

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

See embedding ipython for more detail.

13.9 Debugging the build

We use Meson build system, that you will generally use via the meson-python frontend.

Meson-Python is the wrapper that causes a *pip* command to further call Meson to build Nipy files ready for import.

This can be a problem when you call a command like *pip install* . in the Nipy root directory, and get an obscure error message. It can be difficult to work out where the build failed.

13.9.1 Debug for build failure

To debug builds, drop out of the Meson-Python frontend by invoking Meson directly.

First make sure you have Meson installed, along with its build backend Ninja:

```
pip install meson ninja
```

You may also need Cython>=3:

```
pip install "cython>=3"
```

From the Nipy repository root directory (containing the *pyproject.toml* file):

```
meson setup build
```

This will configure the Meson build in a new subdirectory `build`.

Then:

```
cd build
ninja -j1
```

This will set off the build with a single thread (*-j1*). Prefer a single thread so you get a sequential build. This means that you will see each step running in turn, and you will get any error message at the end of the output. Conversely, if you run with multiple threads (the default), then you'll see warnings and similar from multiple threads, and it will be more difficult to spot the error message among the other outputs.

13.10 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 ;)
```

13.11 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

13.12 A guide to making a nipy release

A guide for developers who are doing a nipy release

13.12.1 Release checklist

- Review the open list of nipy issues. Check whether there are outstanding issues that can be closed, and whether there are any issues that should delay the release. Label them !
- Review and update the release notes. Review and update the `Changelog` file. Get a partial list of contributors with something like:

  ```
  PREV_RELEASE=0.5.0
  git log $PREV_RELEASE.. | grep '^Author' | cut -d' ' -f 2- | sort | uniq
  ```

  where `0.5.0` was the last release tag name.

  Then manually go over `git shortlog $PREV_RELEASE..` to make sure the release notes are as complete as possible and that every contributor was recognized.

- Use the opportunity to update the `.mailmap` file if there are any duplicate authors listed from `git shortlog -ns`.
- Add any new authors to the `AUTHOR` file. Add any new entries to the `THANKS` file.
- Check the copyright years in `doc/conf.py` and `LICENSE`
- Check the output of:

  ```
  rst2html.py README.rst > ~/tmp/readme.html
  ```

  because this will be the output used by PyPI

- Check the dependencies listed in `pyproject.toml` and in `requirements.txt` and in `doc/users/installation.rst`. They should at least match. Do they still hold? Make sure `.github/workflows` is testing these minimum dependencies specifically.
- Check the examples. First download the example data by running something like:

  ```
  # Install data packages.
  pip install https://nipy.org/data-packages/nipy-templates-0.3.tar.gz
  pip install https://nipy.org/data-packages/nipy-data-0.3.tar.gz
  ```

  Then run the tests on the examples with:

```
# Move out of the source directory.
cd ..
# Make log file directory.
mkdir ~/tmp/eg_logs
./nipy/tools/run_log_examples.py nipy/examples --log-path=~/tmp/eg_logs
```

in a virtualenv. Review the output in (e.g.) `~/tmp/eg_logs`. The output file `summary.txt` will have the pass file printout that the `run_log_examples.py` script puts onto stdout while running.

- Check the documentation doctests pass:

```
virtualenv venv
venv/bin/activate
pip install -r doc-requirements.txt
pip install -e .
(cd docs && make clean-doctest)
```

- Check the doc build:

```
virtualenv venv
venv/bin/activate
pip install -r doc-requirements.txt
pip install -e .
(cd docs && make html)
```

- Build and test the Nipy wheels. See the wheel builder README for instructions. In summary, clone the wheel-building repo, edit the `.github/workflow` text files (if present) with the branch or commit for the release, commit and then push back up to github. This will trigger a wheel build and test on macOS, Linux and Windows. Check the build has passed on on the Github interface at https://travis-ci.org/MacPython/nipy-wheels. You'll need commit privileges to the `nipy-wheels` repo; ask Matthew Brett or on the mailing list if you do not have them.

13.12.2 Doing the release

- The release should now be ready.
- Edit `nipy/__init__.py` to set `__version__` to e.g. `0.6.0`.

  Edit `meson.build` to set `version` to match.

  Commit, then:

```
make source-release
```

- For the wheel build / upload, follow the wheel builder README instructions again. Push. Check the build has passed on the Github interface. Now follow the instructions in the page above to download the built wheels to a local machine and upload to PyPI.
- Once everything looks good, you are ready to upload the source release to PyPI. See **`setuptools intro`_**. Make sure you have a file `\$HOME/.pypirc`, of form:

```
[pypi]
username = __token__
```

- Sign and upload the source release to PyPI using Twine:

```
gpg --detach-sign -a dist/nipy*.tar.gz
twine upload dist/nipy*.tar.gz*
```

- Tag the release with tag of form `0.6.0`. *-s* below makes a signed tag:

```
git tag -s 'Second main release' 0.6.0
```

- Now the version number is OK, push the docs to github pages with:

```
make upload-html
```

- Start the new series.

  Edit `nipy/__init__.py` and set version number to something of form:

```
__version__ = "0.6.1.dev1"
```

  where `0.6.0` was the previous release.

- Push tags:

```
git push --tags
```

- Announce to the mailing lists.

13.13 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 FOURTEEN

DEVELOPMENT PLANNING

14.1 Nipy roadmap

We plan to release a prototype 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.

14.2 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…

14.2.1 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.

14.2.2 Bugs

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

- 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*; preliminary search with `grin "class +[a-zA-Z0-9]+ *:"`

14.2.3 Refactorings

- 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?
- 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.

14.2.4 Code Design Thoughts

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

14.2.5 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

FIFTEEN

CODE DISCUSSIONS

These are some developer discussions about design of code in NIPY.

15.1 Understanding voxel and real world mappings

15.1.1 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 positive values for superior (towards the highest point of the head, when standing)

15.1.2 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.

15.2 Image index ordering

15.2.1 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]
```

15.2.2 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]
```

15.2.3 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.

15.2.4 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.

15.3 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)
```

15.3.1 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())
```

15.3.2 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.

15.3.3 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.

15.4 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, adherence 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).

15.5 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

15.5.1 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.

15.5.2 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.

15.5.3 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 access 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 access 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).

15.6 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:
```

(continues on next page)

(continued from previous page)

```
    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.

15.7 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()
```

15.8 Simple image viewer

15.9 Other attempts

http://biomag.wikidot.com/mri-tools http://code.google.com/p/dicompyler https://cirl.berkeley.edu/svn/cburns/trunk/nifti_viewer

15.10 Defining use cases

15.10.1 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*.

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 / coordinate 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)
```

(continues on next page)

(continued from previous page)

```
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 transform* 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 transformation 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.

15.10.2 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*

Taking a mean over a 4D image

We could do this much more simply than below, this is just an example of reducing over a particular axis:

```python
# take mean of 4D image
from glob import glob
import numpy as np
import nipy as ni

fname = 'some4d.nii'

img_list = ni.load_list(fname, axis=3)
vol0 = img_list[0]
arr = vol0.array[:]
for vol in img_list[1:]:
   arr += vol.array
mean_img = ni.Image(arr, vol0.coordmap)
ni.save(mean_img, 'mean_some4d.nii')
```

Taking mean over series of 3D images

Just to show how this works with a list of images:

```
# take mean of some PCA volumes
fnames = glob('some3d*.nii')
vol0 = ni.load(fnames[0])
arr = vol0.array[:]
for fname in fnames[1:]:
    vol = ni.load(fname)
    arr += vol.array
mean_img = ni.Image(arr, vol0.coordmap)
ni.save(mean_img, 'mean_some3ds.nii')
```

Simple motion correction

This is an example of how the 4D -> list of 3D interface works:

```
# motion correction
img_list = ni.load_list(fname, axis=3)
reggie = ni.interfaces.fsl.Register(tol=0.1)
vol0 = img_list[0]
mocod = [] # unresliced
rmocod = [] # resliced
for vol in img_list[1:]:
    rcoord_map = reggie.run(moving=vol, fixed=vol0)
    cmap = ni.ref.compose(rcoord_map, vol.coordmap)
    mocovol = ni.Image(vol.array, cmap)
    # But...
    try:
       a_vol = ni.Image(vol.array, rcoord_map)
    except CoordmapError, msg
       assert msg == 'need coordmap with voxel input'
    mocod.append(mocovol)
    rmocovol = ni.reslice(mocovol, vol0)
    rmocod.append(rmocovol)
rmocod_img = ni.list_to_image(rmocovol)
ni.save(rmocod_img, 'rsome4d.nii')
try:
    mocod_img = ni.list_to_image(mocovol)
except ImageListError:
    print 'That is what I thought; the transforms were not the same'
```

Slice timing

Here putting 3D image into an image list, and back into a 4D image / array:

```
# slice timing
img_list = ni.load_list(fname, axis=2)
slicetimer = ni.interfaces.fsl.SliceTime(algorithm='linear')
vol0 = img_list[0]
try:
   vol0.timestamp
except AttributeError:
   print 'we do not have a timestamp'
try:
   vol0.slicetimes
except AttributeError:
   print 'we do not have slicetimes'
try:
   st_list = slicetimer.run(img)
except SliceTimeError, msg:
   assert msg == 'no timestamp for volume'
TR = 2.0
slicetime = 0.15
sliceaxis = 2
nslices = vol0.array.shape[sliceaxis]
slicetimes = np.range(nslices) * slicetime
timestamps = range(len(img_list)) * TR
# Either the images are in a simple list
for i, img in enumerate(img_list):
   img.timestamp = timestamps[i]
   img.slicetimes = slicetimes
   img.axis['slice'] = sliceaxis # note setting of voxel axis meaning
# if the sliceaxes do not match, error when run
img_list[0].axis['slice'] = 1
try:
   st_list = slicetimer.run(img)
except SliceTimeError, msg:
   assert msg == 'images do not have the same sliceaxes']
# Or - with ImageList object
img_list.timestamps = timestamps
img_list.slicetimes = slicetimes
img_list.axis['slice'] = sliceaxis
# Either way, we run and save
st_list = slicetimer.run(img)
ni.save(ni.list_to_image(st_img), 'stsome4d.nii')
```

Creating an image given data and affine

Showing how we would like the image creation API to look:

```
# making an image from an affine
data = img.array
affine = np.eye(4)
scanner_img = ni.Image(data, ni.ref.voxel2scanner(affine))
mni_img = ni.Image(data, ni.ref.voxel2mni(affine))
```

Coregistration / normalization

Demonstrating coordinate maps and non-linear resampling:

```
# coregistration and normalization
anat_img = ni.load_image('anatomical.nii')
func_img = ni.load_image('epi4d.nii')
template = ni.load_image('mni152T1.nii')

# coreg
coreger = ni.interfaces.fsl.flirt(tol=0.2)
coreg_cmap = coreger.run(fixed=func_img, moving=anat_img)
c_anat_img = ni.Image(anat_img.data, coreg_cmap.compose_with(anat_img.cmap))

# calculate normalization parameters
template_cmap = template.coordmap
template_dims = template.data.shape
c_anat_cmap = c_anat_img.coordmap
normalizer = ni.interfaces.fsl.fnirt(param=3)
norm_cmap = normalizer.run(moving=template, fixed=c_anat_img)

# resample anatomical using calculated coordinate map
full_cmap = norm_cmap.composed_with(template_cmap)
w_anat_data = img.resliced_to_grid(full_cmap, template_dims)
w_anat_img = ni.Image(w_anat_data, template.coordmap)

# resample functionals with calculated coordinate map
w_func_list = []
for img in ni.image_list(func_img, axis=3):
  w_img_data = img.resliced_to_grid(full_cmap, template_dims)
  w_func_list.append(ni.Image(w_img_data, template_cmap))
ni.save(ni.list_to_image(w_func_list), 'stsome4d.nii')
```

15.10.3 Resampling use cases

Use cases for image resampling. See also images.

15.10.4 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.

15.11 Defining use cases

15.11.1 Refactoring imagelists

Usecases for ImageList

Thus far only used in anger in *nipy.modalities.fmri.fmristat.model*, similarly in *nipy.modalities.fmri.spm.model*.

From that file, an object `obj` of class `FmriImageList` must:

- return 4D array from `np.asarray(obj)`, such that the first axis (axis 0) is the axis over which the model is applied
- be indexable such that `obj[0]` returns an Image instance, with valid `shape` and `coordmap` attributes for a time-point 3D volume in the 4D time-series.
- have an attribute `volume_start_times` giving times of the start of each of the volumes in the 4D time series.
- Return the number of volumes in the time-series from `len(obj)`

15.12 Software Design

15.12.1 VTK datasets

Here we describe the VTK dataset model, because of some parallels with our own idea of an image object. The document is from the VTK book - [VTK4]

See also:

- http://code.enthought.com/projects/mayavi/docs/development/html/mayavi/data.html#vtk-data-structures
- http://code.enthought.com/projects/mayavi/docs/development/html/mayavi/auto/example_datasets.html
- http://www.vtk.org/Wiki/VTK/Writing_VTK_files_using_python
- http://www.vtk.org/VTK/img/file-formats.pdf
- https://svn.enthought.com/enthought/attachment/wiki/MayaVi/tvtk_datasets.pdf?format=raw
- http://public.kitware.com/cgi-bin/viewcvs.cgi/*checkout*/Examples/DataManipulation/Python/BuildUGrid.py?root=VTK&content type=text/plain

What is a VTK dataset?

VTK datasets represent discrete spatial data.

Datasets consist of two components:

- *organizing structure* - the topology and geometry
- ***data attributes* - data that can be attached to the topology /**
    geometry above.

Structure: topology / geometry

The structure part of a dataset is the part that gives the position and connection of points in 3D space.

Let us first import *vtk* for our code examples.

```
>>> import vtk
```

An *id* is an index into a given vector

We introduce *id* to explain the code below. An id is simply an index into a vector, and is therefore an integer. Of course the id identifies the element in the vector; as long as you know which vector the id refers to, you can identify the element.

```
>>> pts = vtk.vtkPoints()
>>> id = pts.InsertNextPoint(0, 0, 0)
>>> id == 0
True
>>> id = pts.InsertNextPoint(0, 1, 0)
>>> id == 1
True
>>> pts.GetPoint(1) == (0.0, 1.0, 0.0)
True
```

A dataset has one or more points

Points have coordinates in 3 dimensions, in the order `x`, `y`, `z` - see http://www.vtk.org/doc/release/5.4/html/a00374.html - `GetPoint()`

```
>>> pts = vtk.vtkPoints()
>>> pts.InsertNextPoint(0, 0) # needs 3 coordinates
Traceback (most recent call last):
   ...
TypeError: function takes exactly 3 arguments (2 given)
>>> _ = pts.InsertNextPoint(0, 0, 0) # returns point index in point array
>>> pts.GetPoint(0)
(0.0, 0.0, 0.0)
>>> _ = pts.InsertNextPoint(0, 1, 0)
>>> _ = pts.InsertNextPoint(0, 0, 1)
```

A dataset has one or more cells

A cell is a local specification of the connection between points - an atom of topology in VTK. A cell has a type, and a list of point ids. The point type determines (by convention) what the connectivity of the list of points should be. For example we can make a cell of type `vtkTriangle`. The first point starts the triangle, the next point is the next point in the triangle counterclockwise, connected to the first and third, and the third is the remaining point, connected to the first and second.

```
>>> VTK_TRIANGLE = 5 # A VTK constant identifying the triangle type
>>> triangle = vtk.vtkTriangle()
>>> isinstance(triangle, vtk.vtkCell)
True
>>> triangle.GetCellType() == VTK_TRIANGLE
True
>>> pt_ids = triangle.GetPointIds() # these are default (zeros) at the moment
>>> [pt_ids.GetId(i) for i in range(pt_ids.GetNumberOfIds())] == [0, 0, 0]
True
```

Here we set the ids. The ids refer to the points above. The system does not know this yet, but it will because, later, we are going to associate this cell with the points, in a dataset object.

```
>>> for i in range(pt_ids.GetNumberOfIds()): pt_ids.SetId(i, i)
```

Associating points and cells

We make the most general possible of VTK datasets - the unstructured grid.

```
>>> ugrid = vtk.vtkUnstructuredGrid()
>>> ugrid.Allocate(1, 1)
>>> ugrid.SetPoints(pts)
>>> id = ugrid.InsertNextCell(VTK_TRIANGLE, pt_ids)
```

Data attributes

So far we have specified a triangle, with 3 points, but no associated data.

You can associate data with cells, or with points, or both. Point data associates values (e.g. scalars) with the points in the dataset. Cell data associates values (e.g. scalars) with the cells - in this case one (e.g) scalar value with the whole triangle.

```
>>> pt_data = ugrid.GetPointData()
>>> cell_data = ugrid.GetCellData()
```

There are many data attributes that can be set, include scalars, vectors, normals (normalized vectors), texture coordinates and tensors, using (respectively) `{pt|cell|_data.{Get|Set}{Scalars|Vectors|Normals|TCoords|Tensors}`. For example:

```
>>> pt_data.GetScalars() is None
True
```

But we can set the scalar (or other) data:

```
>>> tri_pt_data = vtk.vtkFloatArray()
>>> for i in range(3): _ = tri_pt_data.InsertNextValue(i)
>>> _ = pt_data.SetScalars(tri_pt_data)
```

To the cells as well, or instead, if we want. Don't forget there is only one cell.

```
>>> tri_cell_data = vtk.vtkFloatArray()
>>> _ = tri_cell_data.InsertNextValue(3)
>>> _ = cell_data.SetScalars(tri_cell_data)
```

You can set different types of data into the same dataset:

```
>>> tri_pt_vecs = vtk.vtkFloatArray()
>>> tri_pt_vecs.SetNumberOfComponents(3)
>>> tri_pt_vecs.InsertNextTuple3(1, 1, 1)
>>> tri_pt_vecs.InsertNextTuple3(2, 2, 2)
>>> tri_pt_vecs.InsertNextTuple3(3, 3, 3)
>>> _ = pt_data.SetVectors(tri_pt_vecs)
```

If you want to look at what you have, run this code

```
# ..testcode:: when live
# make a dataset mapper and actor for our unstructured grid
mapper = vtk.vtkDataSetMapper()
mapper.SetInput(ugrid)
actor = vtk.vtkActor()
actor.SetMapper(mapper)
# Create the usual rendering stuff.
ren = vtk.vtkRenderer()
renWin = vtk.vtkRenderWindow()
renWin.AddRenderer(ren)
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)
# add the actor
ren.AddActor(actor)
# Render the scene and start interaction.
iren.Initialize()
renWin.Render()
iren.Start()
```

CHAPTER

SIXTEEN

DEVELOPER TOOLS

16.1 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.

16.1.1 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 updating 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
```

Note: On older Debian-based releases, the default `M-x rst-compile` command uses `rst2html.py` whereas the command installed is `rst2html`. Symlink was required as a quick fix.

16.1.2 doctest mode

This useful mode for writing doctests (`doctest-mode.el`) cames with `python-mode` package on Debian-based systems. Otherwise see doctest-mode project page.

16.1.3 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` on Debian-based systems) or from the pychecker webpage.

pylint

Install pylint. Debian 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 python3

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.

16.1.4 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.

16.1.5 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)
```

16.1.6 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`).

16.2 Setting up virtualenv

Contents

16.2.1 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)

16.2.2 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.

16.2.3 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
```

16.2.4 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
```

16.2.5 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
```

16.2.6 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.

16.2.7 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
```

(continues on next page)

(continued from previous page)

```
# 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
```

16.2.8 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
```

Part IV

FAQ

CHAPTER SEVENTEEN

WHY …

17.1 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 software.
- 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.

17.2 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

EIGHTEEN

LICENSING

18.1 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.

18.2 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 johns-bsd-pitch, and we agree. Richard Stallman makes the case for the GPL here: http://www.gnu.org/licenses/why-not-lgpl.html

18.3 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.

18.4 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

NINETEEN

DOCUMENTATION FAQ

19.1 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.

19.2 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/nipy
  api/generated/nipy.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.

19.3 Sphinx and reST gotchas

19.3.1 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 V

API

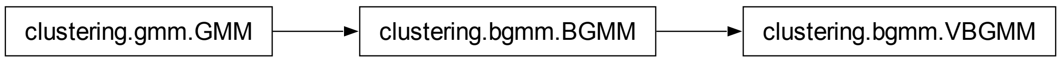
CHAPTER

TWENTY

ALGORITHMS.CLUSTERING.BGMM

20.1 Module: `algorithms.clustering.bgmm`

Inheritance diagram for `nipy.algorithms.clustering.bgmm`:



Bayesian Gaussian Mixture Model Classes: contains the basic fields and methods of Bayesian GMMs the high level functions are/should be binded in C

The base class BGMM relies on an implementation that performs Gibbs sampling

A derived class VBGMM uses Variational Bayes inference instead

A third class is introduces to take advnatge of the old C-bindings, but it is limited to diagonal covariance models

Author : Bertrand Thirion, 2008-2011

20.2 Classes

20.2.1 `BGMM`

class `nipy.algorithms.clustering.bgmm.`**`BGMM`**(*k=1*, *dim=1*, *means=None*, *precisions=None*, *weights=None*, *shrinkage=None*, *dof=None*)

Bases: *`GMM`*

This class implements Bayesian GMMs

this class contains the following fields k: int,

the number of components in the mixture

dim: int,
the dimension of the data

means: array of shape (k, dim)
all the means of the components

precisions: array of shape (k, dim, dim)
the precisions of the components

weights: array of shape (k):
weights of the mixture

shrinkage: array of shape (k):
scaling factor of the posterior precisions on the mean

dof: array of shape (k)
the degrees of freedom of the components

prior_means: array of shape (k, dim):
the prior on the components means

prior_scale: 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_shrinkage: array of shape (k):
scaling factor of the prior precisions on the mean

prior_weights: array of shape (k)
the prior on the components weights

shrinkage: array of shape (k):
scaling factor of the posterior precisions on the mean

dof : array of shape (k): the posterior dofs

`__init__`(*k=1*, *dim=1*, *means=None*, *precisions=None*, *weights=None*, *shrinkage=None*, *dof=None*)
Initialize the structure with the dimensions of the problem Eventually provide different terms

`average_log_like`(*x*, *tiny=1e-15*)
returns the averaged log-likelihood of the mode for the dataset x

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

tiny = 1.e-15: a small constant to avoid numerical singularities

`bayes_factor`(*x*, *z*, *nperm=0*, *verbose=0*)
Evaluate the Bayes Factor of the current model using Chib’s method

Parameters

x: array of shape (nb_samples,dim)
the data from which bic is computed

z: array of shape (nb_samples), type = np.int_
the corresponding classification

nperm=0: int
the number of permutations to sample to model the label switching issue in the computation of the Bayes Factor By default, exhaustive permutations are used

verbose=0: verbosity mode

Returns

bf (float) the computed evidence (Bayes factor)

Notes

See: Marginal Likelihood from the Gibbs Output Journal article by Siddhartha Chib; Journal of the American Statistical Association, Vol. 90, 1995

`bic`(*like*, *tiny=1e-15*)

Computation of bic approximation of evidence

Parameters

like, array of shape (n_samples, self.k)
component-wise likelihood

tiny=1.e-15, a small constant to avoid numerical singularities

Returns

the bic value, float

`check`()

Checking the shape of sifferent matrices involved in the model

`check_x`(*x*)

essentially check that x.shape[1]==self.dim

x is returned with possibly reshaping

`conditional_posterior_proba`(*x*, *z*, *perm=None*)

Compute the probability of the current parameters of self given x and z

Parameters

x: array of shape (nb_samples, dim),
the data from which bic is computed

z: array of shape (nb_samples), type = np.int_,
the corresponding classification

perm: array ok shape(nperm, self.k),typ=np.int_, optional
all permutation of z under which things will be recomputed By default, no permutation is performed

`estimate`(*x*, *niter=100*, *delta=0.0001*, *verbose=0*)

Estimation of the model given a dataset x

Parameters

x array of shape (n_samples,dim)
the data from which the model is estimated

niter=100: maximal number of iterations in the estimation process
delta = 1.e-4: increment of data likelihood at which
convergence is declared

verbose=0: verbosity mode

Returns

bic
[an asymptotic approximation of model evidence]

`evidence`(*x*, *z*, *nperm=0*, *verbose=0*)

See bayes_factor(self, x, z, nperm=0, verbose=0)

`guess_priors`(*x*, *nocheck=0*)

Set the priors in order of having them weakly uninformative this is from Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x, array of shape (nb_samples,self.dim)
the data used in the estimation process

nocheck: boolean, optional,
if nocheck==True, check is skipped

`guess_regularizing`(*x*, *bcheck=1*)

Set the regularizing priors as weakly informative according to Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x array of shape (n_samples,dim)
the data used in the estimation process

`initialize`(*x*)

initialize z using a k-means algorithm, then update the parameters

Parameters

x: array of shape (nb_samples,self.dim)
the data used in the estimation process

`initialize_and_estimate`(*x*, *z=None*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)

Estimation of self given x

Parameters

x array of shape (n_samples,dim)
the data from which the model is estimated

z = None: array of shape (n_samples)
a prior labelling of the data to initialize the computation

niter=100: maximal number of iterations in the estimation process
delta = 1.e-4: increment of data likelihood at which
convergence is declared

ninit=1: number of initialization performed
to reach a good solution

verbose=0: verbosity mode

Returns

the best model is returned

`likelihood`(*x*)

return the likelihood of the model for the data x the values are weighted by the components weights

Parameters

x array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
component-wise likelihood

`map_label`(*x*, *like=None*)

return the MAP labelling of x

Parameters

x array of shape (n_samples,dim)
the data under study

like=None array of shape(n_samples,self.k)
component-wise likelihood if like==None, it is recomputed

Returns

z: array of shape(n_samples): the resulting MAP labelling
of the rows of x

`mixture_likelihood`(*x*)

Returns the likelihood of the mixture for x

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

`plugin`(*means*, *precisions*, *weights*)

Set manually the weights, means and precision of the model

Parameters

means: array of shape (self.k,self.dim)
precisions: array of shape (self.k,self.dim,self.dim)
or (self.k, self.dim)

weights: array of shape (self.k)

`pop`(*z*)

compute the population, i.e. the statistics of allocation

Parameters

z array of shape (nb_samples), type = np.int_
the allocation variable

Returns

hist
[array shape (self.k) count variable]

`probability_under_prior`()

Compute the probability of the current parameters of self given the priors

`sample`(*x*, *niter=1*, *mem=0*, *verbose=0*)

sample the indicator and parameters

Parameters

x array of shape (nb_samples,self.dim)
the data used in the estimation process

niter=1
[the number of iterations to perform]

mem=0: if mem, the best values of the parameters are computed
verbose=0: verbosity mode

Returns

best_weights: array of shape (self.k)
best_means: array of shape (self.k, self.dim)
best_precisions: array of shape (self.k, self.dim, self.dim)
possibleZ: array of shape (nb_samples, niter)
the z that give the highest posterior to the data is returned first

`sample_and_average`(*x*, *niter=1*, *verbose=0*)
sample the indicator and parameters the average values for weights,means, precisions are returned

Parameters

x = array of shape (nb_samples,dim)
the data from which bic is computed

niter=1: number of iterations

Returns

weights: array of shape (self.k)
means: array of shape (self.k,self.dim)
precisions: array of shape (self.k,self.dim,self.dim)
or (self.k, self.dim) these are the average parameters across samplings

Notes

All this makes sense only if no label switching as occurred so this is wrong in general (asymptotically).

fix: implement a permutation procedure for components identification

`sample_indicator`(*like*)
sample the indicator from the likelihood

Parameters

like: array of shape (nb_samples,self.k)
component-wise likelihood

Returns

z: array of shape(nb_samples): a draw of the membership variable

`set_priors`(*prior_means*, *prior_weights*, *prior_scale*, *prior_dof*, *prior_shrinkage*)
Set the prior of the BGMM

Parameters

prior_means: array of shape (self.k,self.dim)
prior_weights: array of shape (self.k)
prior_scale: array of shape (self.k,self.dim,self.dim)
prior_dof: array of shape (self.k)
prior_shrinkage: array of shape (self.k)

`show`(*x*, *gd*, *density=None*, *axes=None*)

Function to plot a GMM, still in progress Currently, works only in 1D and 2D

Parameters

x: array of shape(n_samples, dim)
the data under study

gd: GridDescriptor instance
density: array os shape(prod(gd.n_bins))
density of the model one the discrete grid implied by gd by default, this is recomputed

`show_components`(*x*, *gd*, *density=None*, *mpaxes=None*)

Function to plot a GMM – Currently, works only in 1D

Parameters

x: array of shape(n_samples, dim)
the data under study

gd: GridDescriptor instance
density: array os shape(prod(gd.n_bins))
density of the model one the discrete grid implied by gd by default, this is recomputed

mpaxes: axes handle to make the figure, optional,
if None, a new figure is created

`test`(*x*, *tiny=1e-15*)

Returns the log-likelihood of the mixture for x

Parameters

x array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

ll: array of shape(n_samples)
the log-likelihood of the rows of x

`train`(*x*, *z=None*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)

Idem initialize_and_estimate

`unweighted_likelihood`(*x*)

return the likelihood of each data for each component the values are not weighted by the component weights

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
unweighted component-wise likelihood

Notes

Hopefully faster

`unweighted_likelihood_`(*x*)

return the likelihood of each data for each component the values are not weighted by the component weights

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
unweighted component-wise likelihood

`update`(*x*, *z*)

update function (draw a sample of the GMM parameters)

Parameters

x array of shape (nb_samples,self.dim)
the data used in the estimation process

z array of shape (nb_samples), type = np.int_
the corresponding classification

`update_means`(*x*, *z*)

Given the allocation vector z, and the corresponding data x, resample the mean

Parameters

x: array of shape (nb_samples,self.dim)
the data used in the estimation process

z: array of shape (nb_samples), type = np.int_
the corresponding classification

`update_precisions`(*x*, *z*)

Given the allocation vector z, and the corresponding data x, resample the precisions

Parameters

x array of shape (nb_samples,self.dim)
the data used in the estimation process

z array of shape (nb_samples), type = np.int_
the corresponding classification

`update_weights`(*z*)

Given the allocation vector z, resample the weights parameter

Parameters

z array of shape (nb_samples), type = np.int_
the allocation variable

20.2.2 VBGMM

class `nipy.algorithms.clustering.bgmm.`**`VBGMM`**(*k=1*, *dim=1*, *means=None*, *precisions=None*, *weights=None*, *shrinkage=None*, *dof=None*)

Bases: *BGMM*

Subclass of Bayesian GMMs (BGMM) that implements Variational Bayes estimation of the parameters

`__init__`(*k=1*, *dim=1*, *means=None*, *precisions=None*, *weights=None*, *shrinkage=None*, *dof=None*)

Initialize the structure with the dimensions of the problem Eventually provide different terms

`average_log_like`(*x*, *tiny=1e-15*)

returns the averaged log-likelihood of the mode for the dataset x

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

tiny = 1.e-15: a small constant to avoid numerical singularities

`bayes_factor`(*x*, *z*, *nperm=0*, *verbose=0*)

Evaluate the Bayes Factor of the current model using Chib's method

Parameters

x: array of shape (nb_samples,dim)
the data from which bic is computed

z: array of shape (nb_samples), type = np.int_
the corresponding classification

nperm=0: int
the number of permutations to sample to model the label switching issue in the computation of the Bayes Factor By default, exhaustive permutations are used

verbose=0: verbosity mode

Returns

bf (float) the computed evidence (Bayes factor)

Notes

See: Marginal Likelihood from the Gibbs Output Journal article by Siddhartha Chib; Journal of the American Statistical Association, Vol. 90, 1995

`bic`(*like*, *tiny=1e-15*)

Computation of bic approximation of evidence

Parameters

like, array of shape (n_samples, self.k)
component-wise likelihood

tiny=1.e-15, a small constant to avoid numerical singularities

Returns

the bic value, float

`check()`

Checking the shape of sifferent matrices involved in the model

`check_x(x)`

essentially check that x.shape[1]==self.dim

x is returned with possibly reshaping

`conditional_posterior_proba(x, z, perm=None)`

Compute the probability of the current parameters of self given x and z

Parameters

x: array of shape (nb_samples, dim),
the data from which bic is computed

z: array of shape (nb_samples), type = np.int_,
the corresponding classification

perm: array ok shape(nperm, self.k),typ=np.int_, optional
all permutation of z under which things will be recomputed By default, no permutation is performed

`estimate(x, niter=100, delta=0.0001, verbose=0)`

estimation of self given x

Parameters

x array of shape (nb_samples,dim)
the data from which the model is estimated

z = None: array of shape (nb_samples)
a prior labelling of the data to initialize the computation

niter=100: maximal number of iterations in the estimation process
delta = 1.e-4: increment of data likelihood at which
convergence is declared

verbose=0:
verbosity mode

`evidence(x, like=None, verbose=0)`

computation of evidence bound aka free energy

Parameters

x array of shape (nb_samples,dim)
the data from which evidence is computed

like=None: array of shape (nb_samples, self.k), optional
component-wise likelihood If None, it is recomputed

verbose=0: verbosity model

Returns

ev (float) the computed evidence

`guess_priors(x, nocheck=0)`

Set the priors in order of having them weakly uninformative this is from Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x, array of shape (nb_samples,self.dim)
the data used in the estimation process

nocheck: boolean, optional,
if nocheck==True, check is skipped

`guess_regularizing`(*x*, *bcheck=1*)

Set the regularizing priors as weakly informative according to Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x array of shape (n_samples,dim)
the data used in the estimation process

`initialize`(*x*)

initialize z using a k-means algorithm, then update the parameters

Parameters

x: array of shape (nb_samples,self.dim)
the data used in the estimation process

`initialize_and_estimate`(*x*, *z=None*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)

Estimation of self given x

Parameters

x array of shape (n_samples,dim)
the data from which the model is estimated

z = None: array of shape (n_samples)
a prior labelling of the data to initialize the computation

niter=100: maximal number of iterations in the estimation process
delta = 1.e-4: increment of data likelihood at which
convergence is declared

ninit=1: number of initialization performed
to reach a good solution

verbose=0: verbosity mode

Returns

the best model is returned

`likelihood`(*x*)

return the likelihood of the model for the data x the values are weighted by the components weights

Parameters

x: array of shape (nb_samples, self.dim)
the data used in the estimation process

Returns

like: array of shape(nb_samples, self.k)
component-wise likelihood

`map_label`(*x*, *like=None*)

return the MAP labelling of x

Parameters

x array of shape (nb_samples,dim)
the data under study

like=None array of shape(nb_samples,self.k)
component-wise likelihood if like==None, it is recomputed

Returns

z: array of shape(nb_samples): the resulting MAP labelling
of the rows of x

`mixture_likelihood`(*x*)

Returns the likelihood of the mixture for x

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

`plugin`(*means*, *precisions*, *weights*)

Set manually the weights, means and precision of the model

Parameters

means: array of shape (self.k,self.dim)
precisions: array of shape (self.k,self.dim,self.dim)
or (self.k, self.dim)

weights: array of shape (self.k)

`pop`(*like*, *tiny=1e-15*)

compute the population, i.e. the statistics of allocation

Parameters

like array of shape (nb_samples, self.k):
the likelihood of each item being in each class

`probability_under_prior`()

Compute the probability of the current parameters of self given the priors

`sample`(*x*, *niter=1*, *mem=0*, *verbose=0*)

sample the indicator and parameters

Parameters

x array of shape (nb_samples,self.dim)
the data used in the estimation process

niter=1
[the number of iterations to perform]

mem=0: if mem, the best values of the parameters are computed
verbose=0: verbosity mode

Returns

best_weights: array of shape (self.k)
best_means: array of shape (self.k, self.dim)
best_precisions: array of shape (self.k, self.dim, self.dim)
possibleZ: array of shape (nb_samples, niter)
the z that give the highest posterior to the data is returned first

`sample_and_average`(*x*, *niter=1*, *verbose=0*)

sample the indicator and parameters the average values for weights,means, precisions are returned

Parameters

x = array of shape (nb_samples,dim)
the data from which bic is computed

niter=1: number of iterations

Returns

weights: array of shape (self.k)
means: array of shape (self.k,self.dim)
precisions: array of shape (self.k,self.dim,self.dim)
or (self.k, self.dim) these are the average parameters across samplings

Notes

All this makes sense only if no label switching as occurred so this is wrong in general (asymptotically).

fix: implement a permutation procedure for components identification

`sample_indicator`(*like*)

sample the indicator from the likelihood

Parameters

like: array of shape (nb_samples,self.k)
component-wise likelihood

Returns

z: array of shape(nb_samples): a draw of the membership variable

`set_priors`(*prior_means*, *prior_weights*, *prior_scale*, *prior_dof*, *prior_shrinkage*)

Set the prior of the BGMM

Parameters

prior_means: array of shape (self.k,self.dim)
prior_weights: array of shape (self.k)
prior_scale: array of shape (self.k,self.dim,self.dim)
prior_dof: array of shape (self.k)
prior_shrinkage: array of shape (self.k)

`show`(*x*, *gd*, *density=None*, *axes=None*)

Function to plot a GMM, still in progress Currently, works only in 1D and 2D

Parameters

x: array of shape(n_samples, dim)
the data under study

gd: GridDescriptor instance
density: array os shape(prod(gd.n_bins))
density of the model one the discrete grid implied by gd by default, this is recomputed

`show_components`(*x*, *gd*, *density=None*, *mpaxes=None*)

Function to plot a GMM – Currently, works only in 1D

Parameters

x: array of shape(n_samples, dim)
the data under study

gd: GridDescriptor instance
density: array os shape(prod(gd.n_bins))
density of the model one the discrete grid implied by gd by default, this is recomputed

mpaxes: axes handle to make the figure, optional,
if None, a new figure is created

`test`(*x*, *tiny=1e-15*)

Returns the log-likelihood of the mixture for x

Parameters

x array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

ll: array of shape(n_samples)
the log-likelihood of the rows of x

`train`(*x*, *z=None*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)

Idem initialize_and_estimate

`unweighted_likelihood`(*x*)

return the likelihood of each data for each component the values are not weighted by the component weights

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
unweighted component-wise likelihood

Notes

Hopefully faster

`unweighted_likelihood_`(*x*)

return the likelihood of each data for each component the values are not weighted by the component weights

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
unweighted component-wise likelihood

`update`(*x*, *z*)

update function (draw a sample of the GMM parameters)

Parameters

x array of shape (nb_samples,self.dim)
the data used in the estimation process

z array of shape (nb_samples), type = np.int_
the corresponding classification

`update_means`(*x*, *z*)

Given the allocation vector z, and the corresponding data x, resample the mean

Parameters

x: array of shape (nb_samples,self.dim)
the data used in the estimation process

z: array of shape (nb_samples), type = np.int_
the corresponding classification

`update_precisions`(*x*, *z*)

Given the allocation vector z, and the corresponding data x, resample the precisions

Parameters

x array of shape (nb_samples,self.dim)
the data used in the estimation process

z array of shape (nb_samples), type = np.int_
the corresponding classification

`update_weights`(*z*)

Given the allocation vector z, resample the weights parameter

Parameters

z array of shape (nb_samples), type = np.int_
the allocation variable

20.3 Functions

`nipy.algorithms.clustering.bgmm.detsh`(*H*)

Routine for the computation of determinants of symmetric positive matrices

Parameters

H array of shape(n,n)
the input matrix, assumed symmmetric and positive

Returns

dh: float, the determinant

`nipy.algorithms.clustering.bgmm.dirichlet_eval`(*w*, *alpha*)

Evaluate the probability of a certain discrete draw w from the Dirichlet density with parameters alpha

Parameters

w: array of shape (n)
alpha: array of shape (n)

`nipy.algorithms.clustering.bgmm.dkl_dirichlet`(*w1*, *w2*)

Returns the KL divergence between two dirichlet distribution

Parameters

w1: array of shape(n),
the parameters of the first dirichlet density

w2: array of shape(n),
the parameters of the second dirichlet density

`nipy.algorithms.clustering.bgmm.`**`dkl_gaussian`**`(`*m1*, *P1*, *m2*, *P2*`)`

Returns the KL divergence between gausians densities

Parameters

m1: array of shape (n),
the mean parameter of the first density

P1: array of shape(n,n),
the precision parameters of the first density

m2: array of shape (n),
the mean parameter of the second density

P2: array of shape(n,n),
the precision parameters of the second density

`nipy.algorithms.clustering.bgmm.`**`dkl_wishart`**`(`*a1*, *B1*, *a2*, *B2*`)`

returns the KL divergence bteween two Wishart distribution of parameters (a1,B1) and (a2,B2),

Parameters

a1: Float,
degrees of freedom of the first density

B1: array of shape(n,n),
scale matrix of the first density

a2: Float,
degrees of freedom of the second density

B2: array of shape(n,n),
scale matrix of the second density

Returns

dkl: float, the Kullback-Leibler divergence

`nipy.algorithms.clustering.bgmm.`**`generate_Wishart`**`(`*n*, *V*`)`

Generate a sample from Wishart density

Parameters

n: float,
the number of degrees of freedom of the Wishart density

V: array of shape (n,n)
the scale matrix of the Wishart density

Returns

W: array of shape (n,n)
the draw from Wishart density

`nipy.algorithms.clustering.bgmm.`**`generate_normals`**`(`*m*, *P*`)`

Generate a Gaussian sample with mean m and precision P

Parameters

m array of shape n: the mean vector
P array of shape (n,n): the precision matrix

Returns

ng
[array of shape(n): a draw from the gaussian density]

`nipy.algorithms.clustering.bgmm.`**`generate_perm`**`(`*k*, *nperm=100*`)`

returns an array of shape(nbperm, k) representing the permutations of k elements

Parameters

k, int the number of elements to be permuted
nperm=100 the maximal number of permutations
if gamma(k+1)>nperm: only nperm random draws are generated

Returns

p: array of shape(nperm,k): each row is permutation of k

`nipy.algorithms.clustering.bgmm.`**`multinomial`**`(`*probabilities*`)`

Generate samples form a miltivariate distribution

Parameters

probabilities: array of shape (nelements, nclasses):
likelihood of each element belongin to each class each row is assumedt to sum to 1 One sample is draw from each row, resulting in

Returns

z array of shape (nelements): the draws,
that take values in [0..nclasses-1]

`nipy.algorithms.clustering.bgmm.`**`normal_eval`**`(`*mu*, *P*, *x*, *dP=None*`)`

Probability of x under normal(mu, inv(P))

Parameters

mu: array of shape (n),
the mean parameter

P: array of shape (n, n),
the precision matrix

x: array of shape (n),
the data to be evaluated

Returns

(float) the density

`nipy.algorithms.clustering.bgmm.`**`wishart_eval`**`(`*n*, *V*, *W*, *dV=None*, *dW=None*, *piV=None*`)`

Evaluation of the probability of W under Wishart(n,V)

Parameters

n: float,
the number of degrees of freedom (dofs)

V: array of shape (n,n)
the scale matrix of the Wishart density

W: array of shape (n,n)
the sample to be evaluated

dV: float, optional,
determinant of V

dW: float, optional,
determinant of W

piV: array of shape (n,n), optional
inverse of V

Returns

(float) the density

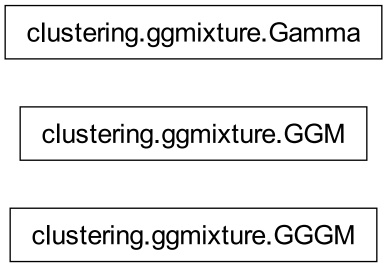
CHAPTER

TWENTYONE

ALGORITHMS.CLUSTERING.GGMIXTURE

21.1 Module: `algorithms.clustering.ggmixture`

Inheritance diagram for `nipy.algorithms.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

21.2 Classes

21.2.1 `GGGM`

class `nipy.algorithms.clustering.ggmixture.`**`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])*)

Bases: `object`

The basic one dimensional Gamma-Gaussian-Gamma Mixture estimation class, where the first 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])*)

Constructor

Parameters

shape_n
[float, optional]

scale_n: float, optional
parameters of the nehative gamma; must be positive

mean
[float, optional]

var
[float, optional] parameters of the gaussian ; var must be positive

shape_p
[float, optional]

scale_p
[float, optional] parameters of the positive gamma; must be positive

mixt
[array of shape (3,), optional] the mixing proportions; they should be positive and sum to 1

`Estep`(*x*)

Update probabilistic memberships of the three components

Parameters

x: array of shape (nbitems,)
the input data

Returns

z: ndarray of shape (nbitems, 3)
probabilistic membership

Notes

z[0,:] is the membership the negative gamma z[1,:] is the membership of the gaussian z[2,:] is the membership of the positive gamma

`Mstep`(*x*, *z*)

Mstep of the estimation: Maximum likelihood update the parameters of the three components

Parameters

x: array of shape (nbitem,)
input data

z: array of shape (nbitems,3)
probabilistic membership

`component_likelihood`(*x*)

Compute the likelihood of the data x under the three components negative gamma, gaussina, positive gaussian

Parameters

x: array of shape (nbitem,)
the data under evaluation

Returns

ng,y,pg: three arrays of shape(nbitem)
The likelihood of the data under the 3 components

`estimate`(*x*, *niter=100*, *delta=0.0001*, *bias=0*, *verbose=0*, *gaussian_mix=0*)

Whole EM estimation procedure:

Parameters

x: array of shape (nbitem)
input data

niter: integer, optional
max number of iterations

delta: float, optional
increment in LL at which convergence is declared

bias: float, optional
lower bound on the gaussian variance (to avoid shrinkage)

gaussian_mix: float, optional
if nonzero, lower bound on the gaussian mixing weight (to avoid shrinkage)

verbose: 0, 1 or 2
verbosity level

Returns

z: array of shape (nbitem, 3)
the membership matrix

`init`(*x*, *mixt=None*)

initialization of the different parameters

Parameters

x: array of shape(nbitems)
the data to be processed

mixt
[None or array of shape(3), optional] prior mixing proportions. If None, the classes have equal weight

`init_fdr`(*x*, *dof=-1*, *copy=True*)

Initialization 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

x: array of shape (nbitem)
the data under consideration

dof: integer, optional
number of degrees of freedom if x is thought to be a Student variate. By default, it is handled as a normal

copy: boolean, optional
If True, copy the data.

`parameters()`

Print the parameters

`posterior(x)`

Compute the posterior probability of the three components given the data

Parameters

x: array of shape (nbitem,)
the data under evaluation

Returns

ng,y,pg: three arrays of shape(nbitem)
the posteriori of the 3 components given the data

Notes

ng + y + pg = np.ones(nbitem)

`show(x, mpaxes=None)`

Visualization of mixture shown on the empirical histogram of x

Parameters

x: ndarray of shape (nditem,)
data

mpaxes: matplotlib axes, optional
axes handle used for the plot if None, new axes are created.

21.2.2 GGM

class `nipy.algorithms.clustering.ggmixture.GGM`(*shape=1*, *scale=1*, *mean=0*, *var=1*, *mixt=0.5*)

Bases: `object`

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 scalar members - mean: gaussian mean - var: gaussian variance (non-negative) - shape: gamma shape (non-negative) - scale: gamma scale (non-negative) - mixt: mixture parameter (non-negative, weight of the gamma)

`__init__`(*shape=1*, *scale=1*, *mean=0*, *var=1*, *mixt=0.5*)

`Estep(x)`

E step of the estimation: Estimation of ata membsership

Parameters

x: array of shape (nbitems,)
input data

Returns

z: array of shape (nbitems, 2)
the membership matrix

`Mstep`(*x*, *z*)

Mstep of the model: maximum likelihood estimation of the parameters of the model

Parameters

x
[array of shape (nbitems,)] input data

z array of shape(nbitrems, 2)
the membership matrix

`estimate`(*x*, *niter=10*, *delta=0.0001*, *verbose=False*)

Complete EM estimation procedure

Parameters

x
[array of shape (nbitems,)] the data to be processed

niter
[int, optional] max nb of iterations

delta
[float, optional] criterion for convergence

verbose
[bool, optional] If True, print values during iterations

Returns

LL, float
average final log-likelihood

`parameters`()

print the parameters of self

`posterior`(*x*)

Posterior probability of observing the data x for each component

Parameters

x: array of shape (nbitems,)
the data to be processed

Returns

y, pg
[arrays of shape (nbitem)] the posterior probability

`show`(*x*)

Visualization of the mm based on the empirical histogram of x

Parameters

x
[array of shape (nbitems,)] the data to be processed

21.2.3 Gamma

class `nipy.algorithms.clustering.ggmixture.`**`Gamma`**(*shape=1*, *scale=1*)

Bases: `object`

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=1e-07*)

ML estimation of the Gamma parameters

`parameters`()

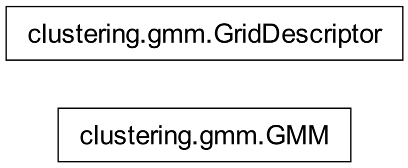
CHAPTER

TWENTYTWO

ALGORITHMS.CLUSTERING.GMM

22.1 Module: `algorithms.clustering.gmm`

Inheritance diagram for `nipy.algorithms.clustering.gmm`:



Gaussian Mixture Model Class: contains the basic fields and methods of GMMs The class GMM _old uses C bindings which are computationally and memory efficient.

Author : Bertrand Thirion, 2006-2009

22.2 Classes

22.2.1 `GMM`

class `nipy.algorithms.clustering.gmm.`**`GMM`**(*k=1*, *dim=1*, *prec_type='full'*, *means=None*, *precisions=None*, *weights=None*)

Bases: `object`

Standard GMM.

this class contains the following members k (int): the number of components in the mixture dim (int): is the dimension of the data prec_type = 'full' (string) is the parameterization

of the precisions/covariance matrices: either 'full' or 'diagonal'.

means: array of shape (k,dim):
all the means (mean parameters) of the components

precisions: array of shape (k,dim,dim):
the precisions (inverse covariance matrix) of the components

weights: array of shape(k): weights of the mixture

`__init__`(*k=1*, *dim=1*, *prec_type='full'*, *means=None*, *precisions=None*, *weights=None*)

Initialize the structure, at least with the dimensions of the problem

Parameters

k (int) the number of classes of the model
dim (int) the dimension of the problem
prec_type = 'full'
[coavriance:precision parameterization] (diagonal 'diag' or full 'full').

means = None: array of shape (self.k,self.dim)
precisions = None: array of shape (self.k,self.dim,self.dim)
or (self.k, self.dim)

weights=None: array of shape (self.k)
By default, means, precision and weights are set as
zeros()
eye()
1/k ones()
with the correct dimensions

`average_log_like`(*x*, *tiny=1e-15*)

returns the averaged log-likelihood of the mode for the dataset x

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

tiny = 1.e-15: a small constant to avoid numerical singularities

`bic`(*like*, *tiny=1e-15*)

Computation of bic approximation of evidence

Parameters

like, array of shape (n_samples, self.k)
component-wise likelihood

tiny=1.e-15, a small constant to avoid numerical singularities

Returns

the bic value, float

`check`()

Checking the shape of different matrices involved in the model

`check_x`(*x*)

essentially check that x.shape[1]==self.dim

x is returned with possibly reshaping

`estimate`(*x*, *niter=100*, *delta=0.0001*, *verbose=0*)

Estimation of the model given a dataset x

Parameters

x array of shape (n_samples,dim)
the data from which the model is estimated

niter=100: maximal number of iterations in the estimation process
delta = 1.e-4: increment of data likelihood at which
convergence is declared

verbose=0: verbosity mode

Returns

bic
[an asymptotic approximation of model evidence]

`evidence`(*x*)

Computation of bic approximation of evidence

Parameters

x array of shape (n_samples,dim)
the data from which bic is computed

Returns

the bic value

`guess_regularizing`(*x*, *bcheck=1*)

Set the regularizing priors as weakly informative according to Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x array of shape (n_samples,dim)
the data used in the estimation process

`initialize`(*x*)

Initializes self according to a certain dataset x: 1. sets the regularizing hyper-parameters 2. initializes z using a k-means algorithm, then 3. update the parameters

Parameters

x, array of shape (n_samples,self.dim)
the data used in the estimation process

`initialize_and_estimate`(*x*, *z=None*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)

Estimation of self given x

Parameters

x array of shape (n_samples,dim)
the data from which the model is estimated

z = None: array of shape (n_samples)
a prior labelling of the data to initialize the computation

niter=100: maximal number of iterations in the estimation process
delta = 1.e-4: increment of data likelihood at which
convergence is declared

ninit=1: number of initialization performed
to reach a good solution

verbose=0: verbosity mode

Returns

the best model is returned

`likelihood`(*x*)

return the likelihood of the model for the data x the values are weighted by the components weights

Parameters

x array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
component-wise likelihood

`map_label`(*x*, *like=None*)

return the MAP labelling of x

Parameters

x array of shape (n_samples,dim)
the data under study

like=None array of shape(n_samples,self.k)
component-wise likelihood if like==None, it is recomputed

Returns

z: array of shape(n_samples): the resulting MAP labelling
of the rows of x

`mixture_likelihood`(*x*)

Returns the likelihood of the mixture for x

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

`plugin`(*means*, *precisions*, *weights*)

Set manually the weights, means and precision of the model

Parameters

means: array of shape (self.k,self.dim)
precisions: array of shape (self.k,self.dim,self.dim)
or (self.k, self.dim)

weights: array of shape (self.k)

`pop`(*like*, *tiny=1e-15*)

compute the population, i.e. the statistics of allocation

Parameters

like: array of shape (n_samples,self.k):
the likelihood of each item being in each class

`show`(*x*, *gd*, *density=None*, *axes=None*)

Function to plot a GMM, still in progress Currently, works only in 1D and 2D

Parameters

x: array of shape(n_samples, dim)
the data under study

gd: GridDescriptor instance
density: array os shape(prod(gd.n_bins))
density of the model one the discrete grid implied by gd by default, this is recomputed

`show_components`(*x*, *gd*, *density=None*, *mpaxes=None*)
Function to plot a GMM – Currently, works only in 1D

Parameters

x: array of shape(n_samples, dim)
the data under study

gd: GridDescriptor instance
density: array os shape(prod(gd.n_bins))
density of the model one the discrete grid implied by gd by default, this is recomputed

mpaxes: axes handle to make the figure, optional,
if None, a new figure is created

`test`(*x*, *tiny=1e-15*)
Returns the log-likelihood of the mixture for x

Parameters

x array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

ll: array of shape(n_samples)
the log-likelihood of the rows of x

`train`(*x*, *z=None*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)
Idem initialize_and_estimate

`unweighted_likelihood`(*x*)
return the likelihood of each data for each component the values are not weighted by the component weights

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
unweighted component-wise likelihood

Notes

Hopefully faster

`unweighted_likelihood_`(*x*)
return the likelihood of each data for each component the values are not weighted by the component weights

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
unweighted component-wise likelihood

`update`(*x*, *l*)
Identical to self._Mstep(x,l)

22.2.2 `GridDescriptor`

class `nipy.algorithms.clustering.gmm.`**`GridDescriptor`**(*dim=1*, *lim=None*, *n_bins=None*)
Bases: `object`

A tiny class to handle cartesian grids

`__init__`(*dim=1*, *lim=None*, *n_bins=None*)

Parameters

dim: int, optional,
the dimension of the grid

lim: list of len(2*self.dim),
the limits of the grid as (xmin, xmax, ymin, ymax, …)

n_bins: list of len(self.dim),
the number of bins in each direction

`make_grid`()
Compute the grid points

Returns

grid: array of shape (nb_nodes, self.dim)
where nb_nodes is the prod of self.n_bins

`set`(*lim*, *n_bins=10*)
set the limits of the grid and the number of bins

Parameters

lim: list of len(2*self.dim),
the limits of the grid as (xmin, xmax, ymin, ymax, …)

n_bins: list of len(self.dim), optional
the number of bins in each direction

22.3 Functions

`nipy.algorithms.clustering.gmm.`**`best_fitting_GMM`**(*x*, *krange*, *prec_type='full'*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)
Given a certain dataset x, find the best-fitting GMM with a number k of classes in a certain range defined by krange

Parameters

x: array of shape (n_samples,dim)
the data from which the model is estimated

krange: list of floats,
the range of values to test for k

prec_type: string (to be chosen within 'full','diag'), optional,
the covariance parameterization

niter: int, optional,
maximal number of iterations in the estimation process

delta: float, optional,
increment of data likelihood at which convergence is declared

ninit: int
number of initialization performed

verbose=0: verbosity mode

Returns

mg
[the best-fitting GMM instance]

`nipy.algorithms.clustering.gmm.plot2D`(*x*, *my_gmm*, *z=None*, *with_dots=True*, *log_scale=False*, *mpaxes=None*, *verbose=0*)

Given a set of points in a plane and a GMM, plot them

Parameters

x: array of shape (npoints, dim=2),
sample points

my_gmm: GMM instance,
whose density has to be plotted

z: array of shape (npoints), optional
that gives a labelling of the points in x by default, it is not taken into account

with_dots, bool, optional
whether to plot the dots or not

log_scale: bool, optional
whether to plot the likelihood in log scale or not

mpaxes=None, int, optional
if not None, axes handle for plotting

verbose: verbosity mode, optional

Returns

gd, GridDescriptor instance,
that represents the grid used in the function

ax, handle to the figure axes

Notes

`my_gmm` is assumed to have have a 'nixture_likelihood' method that takes an array of points of shape (np, dim) and returns an array of shape (np,my_gmm.k) that represents the likelihood component-wise

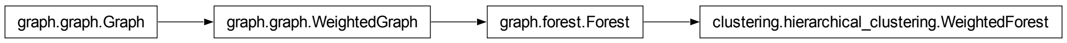
CHAPTER

TWENTYTHREE

ALGORITHMS.CLUSTERING.HIERARCHICAL_CLUSTERING

23.1 Module: `algorithms.clustering.hierarchical_clustering`

Inheritance diagram for `nipy.algorithms.clustering.hierarchical_clustering`:


graph.graph.Graph
graph.graph.WeightedGraph
graph.forest.Forest
clustering.hierarchical_clustering.WeightedForest

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

23.2 Class

23.3 `WeightedForest`

class `nipy.algorithms.clustering.hierarchical_clustering.`**`WeightedForest`**(*V*, *parents=None*, *height=None*)

Bases: `Forest`

This is a weighted Forest structure, i.e. a tree - each 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

`__init__`(*V*, *parents=None*, *height=None*)

Parameters

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

`adjacency()`

returns the adjacency matrix of the graph as a sparse coo matrix

Returns

adj: scipy.sparse matrix instance,
that encodes the adjacency matrix of self

`all_distances(`*seed=None*`)`

returns all the distances of the graph as a tree

Parameters

seed=None array of shape(nbseed) with valuesin [0..self.V-1]
set of vertices from which tehe distances are computed

Returns

dg: array of shape(nseed, self.V), the resulting distances

Notes

By convention infinite distances are given the distance np.inf

`anti_symmeterize()`

anti-symmeterize self, i.e. produces the graph whose adjacency matrix would be the antisymmetric part of its current adjacency matrix

`cc()`

Compte the different connected components of the graph.

Returns

label: array of shape(self.V), labelling of the vertices

`check()`

Check that self is indeed a forest, i.e. contains no loop

Returns

a boolean b=0 iff there are loops, 1 otherwise

Notes

Slow implementation, might be rewritten in C or cython

`check_compatible_height()`

Check that height[parents[i]]>=height[i] for all nodes

`cliques()`

Extraction of the graphe cliques these are defined using replicator dynamics equations

Returns

cliques: array of shape (self.V), type (np.int_)
labelling of the vertices according to the clique they belong to

`compact_neighb()`

returns a compact representation of self

Returns

idx: array of of shape(self.V + 1):
the positions where to find the neighbors of each node within neighb and weights

neighb: array of shape(self.E), concatenated list of neighbors
weights: array of shape(self.E), concatenated list of weights

`compute_children()`

Define the children of each node (stored in self.children)

`copy()`

returns a copy of self

`cut_redundancies()`

Returns a graph with redundant edges removed: ecah edge (ab) is present only once in the edge matrix: the correspondng weights are added.

Returns

the resulting WeightedGraph

`define_graph_attributes()`

define the edge and weights array

`degrees()`

Returns the degree of the graph vertices.

Returns

rdegree: (array, type=int, shape=(self.V,)), the right degrees
ldegree: (array, type=int, shape=(self.V,)), the left degrees

`depth_from_leaves()`

compute an index for each node: 0 for the leaves, 1 for their parents etc. and maximal for the roots.

Returns

depth: array of shape (self.V): the depth values of the vertices

`dijkstra(`*seed=0*`)`

Returns all the [graph] geodesic distances starting from seed x

seed (int, >-1, <self.V) or array of shape(p)
edge(s) from which the distances are computed

Returns

dg: array of shape (self.V),
the graph distance dg from ant vertex to the nearest seed

Notes

It is mandatory that the graph weights are non-negative

`floyd`(*seed=None*)

Compute all the geodesic distances starting from seeds

Parameters

seed= None: array of shape (nbseed), type np.int_
vertex indexes from which the distances are computed if seed==None, then every edge is a seed point

Returns

dg array of shape (nbseed, self.V)
the graph distance dg from each seed to any vertex

Notes

It is mandatory that the graph weights are non-negative. The algorithm proceeds by repeating Dijkstra's algo for each seed. Floyd's algo is not used (O(self.V)^3 complexity…)

`from_3d_grid`(*xyz*, *k=18*)

Sets the graph to be the topological neighbours graph of the three-dimensional coordinates set xyz, in the k-connectivity scheme

Parameters

xyz: array of shape (self.V, 3) and type np.int_,
k = 18: the number of neighbours considered. (6, 18 or 26)

Returns

E(int): the number of edges of self

`get_E`()

To get the number of edges in the graph

`get_V`()

To get the number of vertices in the graph

`get_children`(*v=-1*)

Get the children of a node/each node

Parameters

v: int, optional
a node index

Returns

children: list of int the list of children of node v (if v is provided)
a list of lists of int, the children of all nodes otherwise

`get_descendants`(*v*, *exclude_self=False*)

returns the nodes that are children of v as a list

Parameters

v: int, a node index

Returns

desc: list of int, the list of all descendant of the input node

`get_edges`()

To get the graph's edges

`get_height`()

Get the height array

`get_vertices`()

To get the graph's vertices (as id)

`get_weights`()

`is_connected`()

States whether self is connected or not

`isleaf`()

Identification of the leaves of the forest

Returns

leaves: bool array of shape(self.V), indicator of the forest's leaves

`isroot`()

Returns an indicator of nodes being roots

Returns

roots, array of shape(self.V, bool), indicator of the forest's roots

`kruskal`()

Creates the Minimum Spanning Tree of self using Kruskal's algo. efficient is self is sparse

Returns

K, WeightedGraph instance: the resulting MST

Notes

If self contains several connected components, will have the same number k of connected components

`leaves_of_a_subtree`(*ids*, *custom=False*)

tests whether the given nodes are the leaves of a certain subtree

Parameters

ids: array of shape (n) that takes values in [0..self.V-1]
custom == False, boolean
if custom==true the behavior of the function is more specific - the different connected components are considered as being in a same greater tree - when a node has more than two subbranches, any subset of these children is considered as a subtree

`left_incidence()`

Return left incidence matrix

Returns

left_incid: list
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()`

returns the set of neighbors of self as a list of arrays

`list_of_subtrees()`

returns the list of all non-trivial subtrees in the graph Caveat: this function assumes that the vertices are sorted in a way such that parent[i]>i for all i Only the leaves are listeed, not the subtrees themselves

`main_cc()`

Returns the indexes of the vertices within the main cc

Returns

idx: array of shape (sizeof main cc)

`merge_simple_branches()`

Return a subforest, where chained branches are collapsed

Returns

sf, Forest instance, same as self, without any chain

`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

Parameters

c=0 in {0, 1, 2}, optional: index that designates the way
according to which D is normalized 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

Notes

Note that when sum_{edge[e, .] == a } D[e] = 0, nothing is performed

`partition`(*threshold*)

Partition the tree according to a cut criterion

`plot`(*ax=None*)

Plot the dendrogram associated with self the rank of the data in the dendogram is returned

Parameters

ax: axis handle, optional

Returns

ax, the axis handle

`plot_height()`

Plot the height of the non-leaves nodes

`propagate_upward`(*label*)

Propagation of a certain labelling from leaves to roots Assuming that label is a certain positive integer field this propagates these labels to the parents whenever the children nodes have coherent properties otherwise the parent value is unchanged

Parameters

label: array of shape(self.V)

Returns

label: array of shape(self.V)

`propagate_upward_and`(*prop*)

propagates from leaves to roots some binary property of the nodes so that prop[parents] = logical_and(prop[children])

Parameters

prop, array of shape(self.V), the input property

Returns

prop, array of shape(self.V), the output property field

`remove_edges`(*valid*)

Removes all the edges for which valid==0

Parameters

valid
[(self.E,) array]

`remove_trivial_edges`()

Removes trivial edges, i.e. edges that are (vv)-like self.weights and self.E are corrected accordingly

Returns

self.E (int): The number of edges

`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.

Returns

order: array of shape(self.V)
the order of the old vertices in the reordered graph

`right_incidence`()

Return right incidence matrix

Returns

right_incid: list
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_edges`(*edges*)

Sets the graph's edges

Preconditions:

- edges has a correct size
- edges take values in [1..V]

`set_euclidian`(*X*)

Compute the weights of the graph as the distances between the corresponding rows of X, which represents an embedding of self

Parameters

X array of shape (self.V, edim),
the coordinate matrix of the embedding

`set_gaussian`(*X*, *sigma=0*)

Compute the weights of the graph as a gaussian function of the distance between the corresponding rows of X, which represents an embedding of self

Parameters

X array of shape (self.V, dim)
the coordinate matrix of the embedding

sigma=0, float: the parameter of the gaussian function

Notes

When sigma == 0, the following value is used: `sigma = sqrt(mean(||X[self.edges[:, 0], :]-X[self.edges[:, 1], :]||^2))`

`set_height`(*height=None*)

Set the height array

`set_weights`(*weights*)

Set edge weights

Parameters

weights: array
array shape(self.V): edges weights

`show`(*X=None*, *ax=None*)

Plots the current graph in 2D

Parameters

X
[None or array of shape (self.V, 2)] a set of coordinates that can be used to embed the vertices in 2D. If X.shape[1]>2, a svd reduces X for display. By default, the graph is presented on a circle

ax: None or int, optional
ax handle

Returns

ax: axis handle

Notes

This should be used only for small graphs.

`split(k)`

idem as partition, but a number of components are supplied instead

`subforest(valid)`

Creates a subforest with the vertices for which valid > 0

Parameters

valid: array of shape (self.V): indicator of the selected nodes

Returns

subforest: a new forest instance, with a reduced set of nodes

Notes

The children of deleted vertices become their own parent

`subgraph(valid)`

Creates a subgraph with the vertices for which valid>0 and with the corresponding set of edges

Parameters

valid, array of shape (self.V): nonzero for vertices to be retained

Returns

G, WeightedGraph instance, the desired subgraph of self

Notes

The vertices are renumbered as [1..p] where p = sum(valid>0) when sum(valid==0) then None is returned

`symmeterize()`

Symmeterize self, modify edges and weights so that self.adjacency becomes the symmetric part of the current self.adjacency.

`to_coo_matrix()`

Return adjacency matrix as coo sparse

Returns

sp: scipy.sparse matrix instance
that encodes the adjacency matrix of self

`tree_depth()`

Returns the number of hierarchical levels in the tree

`voronoi_diagram(seeds, samples)`

Defines the graph as the Voronoi diagram (VD) that links the seeds. The VD is defined using the sample points.

Parameters

seeds: array of shape (self.V, dim)
samples: array of shape (nsamples, dim)

Notes

By default, the weights are a Gaussian function of the distance The implementation is not optimal

`voronoi_labelling`(*seed*)

Performs a voronoi labelling of the graph

Parameters

seed: array of shape (nseeds), type (np.int_),
vertices from which the cells are built

Returns

labels: array of shape (self.V) the labelling of the vertices

23.4 Functions

`nipy.algorithms.clustering.hierarchical_clustering.average_link_graph`(*G*)

Agglomerative function based on a (hopefully sparse) similarity graph

Parameters

G the input graph

Returns

t a weightForest structure that represents the dendrogram of the data

`nipy.algorithms.clustering.hierarchical_clustering.average_link_graph_segment`(*G*, *stop=0*, *qmax=1*, *verbose=False*)

Agglomerative function based on a (hopefully sparse) similarity graph

Parameters

G the input graph
stop: float
the stopping criterion

qmax: int, optional
the number of desired clusters (in the limit of the stopping criterion)

verbose
[bool, optional] If True, print diagnostic information

Returns

u: array of shape (G.V)
a labelling of the graph vertices according to the criterion

cost: array of shape (G.V (?))
the cost of each merge step during the clustering procedure

`nipy.algorithms.clustering.hierarchical_clustering.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) `nipy.algorithms.clustering.hierarchical_clustering.`**`ward`**`(`*G*, *feature*, *verbose=False*`)`

Agglomerative function based on a topology-defining graph and a feature matrix.

Parameters

G
[graph] the input graph (a topological graph essentially)

feature
[array of shape (G.V,dim_feature)] vectorial information related to the graph vertices

verbose
[bool, optional] If True, print diagnostic information

Returns

t
[`WeightedForest` instance] structure that represents the dendrogram

Notes

When G has more than 1 connected component, t is no longer a tree. This case is handled cleanly now

`nipy.algorithms.clustering.hierarchical_clustering.`**`ward_field_segment`**`(`*F*, *stop=-1*, *qmax=-1*, *verbose=False*`)`

Agglomerative function based on a field structure

Parameters

F the input field (graph+feature)
stop: float, optional
the stopping crterion. if stop==-1, then no stopping criterion is used

qmax: int, optional
the maximum number of desired clusters (in the limit of the stopping criterion)

verbose
[bool, optional] If True, print diagnostic information

Returns

u: array of shape (F.V)
labelling of the graph vertices according to the criterion

cost array of shape (F.V - 1)
the cost of each merge step during the clustering procedure

Notes

See ward_quick_segment for more information

Caveat : only approximate

`nipy.algorithms.clustering.hierarchical_clustering.`**`ward_quick`**`(`*G*, *feature*, *verbose=False*`)`

Agglomerative function based on a topology-defining graph and a feature matrix.

Parameters

G
[graph instance] topology-defining graph

feature: array of shape (G.V,dim_feature)
some vectorial information related to the graph vertices

verbose
[bool, optional] If True, print diagnostic information

Returns

t: weightForest instance,
that represents the dendrogram of the data

Notes

Hopefully a quicker version
A euclidean distance is used in the feature space
Caveat
[only approximate]

`nipy.algorithms.clustering.hierarchical_clustering.`**`ward_quick_segment`**(*G*, *feature*, *stop=-1*, *qmax=1*, *verbose=False*)

Agglomerative function based on a topology-defining graph and a feature matrix.

Parameters

G: labs.graph.WeightedGraph instance
the input graph (a topological graph essentially)

feature array of shape (G.V,dim_feature)
vectorial information related to the graph vertices

stop1
[int or float, optional] the stopping crterion if stop==-1, then no stopping criterion is used

qmax
[int, optional] the maximum number of desired clusters (in the limit of the stopping criterion)

verbose
[bool, optional] If True, print diagnostic information

Returns

u: array of shape (G.V)
labelling of the graph vertices according to the criterion

cost: array of shape (G.V - 1)
the cost of each merge step during the clustering procedure

Notes

Hopefully a quicker version

A euclidean distance is used in the feature space

Caveat : only approximate

`nipy.algorithms.clustering.hierarchical_clustering.`**`ward_segment`**(*G*, *feature*, *stop=-1*, *qmax=1*, *verbose=False*)

Agglomerative function based on a topology-defining graph and a feature matrix.

Parameters

G
[graph object] the input graph (a topological graph essentially)

feature
[array of shape (G.V,dim_feature)] some vectorial information related to the graph vertices

stop
[int or float, optional] the stopping crterion. if stop==-1, then no stopping criterion is used

qmax
[int, optional] the maximum number of desired clusters (in the limit of the stopping criterion)

verbose
[bool, optional] If True, print diagnostic information

Returns

u: array of shape (G.V):
a labelling of the graph vertices according to the criterion

cost: array of shape (G.V - 1)
the cost of each merge step during the clustering procedure

Notes

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 !

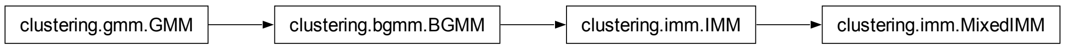
CHAPTER

TWENTYFOUR

ALGORITHMS.CLUSTERING.IMM

24.1 Module: `algorithms.clustering.imm`

Inheritance diagram for `nipy.algorithms.clustering.imm`:



Infinite mixture model : A generalization of Bayesian mixture models with an unspecified number of classes

24.2 Classes

24.2.1 `IMM`

class `nipy.algorithms.clustering.imm.`**`IMM`**(*alpha=0.5*, *dim=1*)

Bases: *BGMM*

The class implements Infinite Gaussian Mixture model or Dirichlet Process Mixture model. This is simply a generalization of Bayesian Gaussian Mixture Models with an unknown number of classes.

`__init__`(*alpha=0.5*, *dim=1*)

Parameters

alpha: float, optional,
the parameter for cluster creation

dim: int, optional,
the dimension of the the data

Note: use the function set_priors() to set adapted priors

`average_log_like`(*x*, *tiny=1e-15*)

returns the averaged log-likelihood of the mode for the dataset x

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

tiny = 1.e-15: a small constant to avoid numerical singularities

`bayes_factor`(*x*, *z*, *nperm=0*, *verbose=0*)

Evaluate the Bayes Factor of the current model using Chib's method

Parameters

x: array of shape (nb_samples,dim)
the data from which bic is computed

z: array of shape (nb_samples), type = np.int_
the corresponding classification

nperm=0: int
the number of permutations to sample to model the label switching issue in the computation of the Bayes Factor By default, exhaustive permutations are used

verbose=0: verbosity mode

Returns

bf (float) the computed evidence (Bayes factor)

Notes

See: Marginal Likelihood from the Gibbs Output Journal article by Siddhartha Chib; Journal of the American Statistical Association, Vol. 90, 1995

`bic`(*like*, *tiny=1e-15*)

Computation of bic approximation of evidence

Parameters

like, array of shape (n_samples, self.k)
component-wise likelihood

tiny=1.e-15, a small constant to avoid numerical singularities

Returns

the bic value, float

`check`()

Checking the shape of sifferent matrices involved in the model

`check_x`(*x*)

essentially check that x.shape[1]==self.dim

x is returned with possibly reshaping

`conditional_posterior_proba`(*x*, *z*, *perm=None*)

Compute the probability of the current parameters of self given x and z

Parameters

x: array of shape (nb_samples, dim),
the data from which bic is computed

z: array of shape (nb_samples), type = np.int_,
the corresponding classification

perm: array ok shape(nperm, self.k),typ=np.int_, optional
all permutation of z under which things will be recomputed By default, no permutation is performed

`cross_validated_update`(*x*, *z*, *plike*, *kfold=10*)

This is a step in the sampling procedure that uses internal corss_validation

Parameters

x: array of shape(n_samples, dim),
the input data

z: array of shape(n_samples),
the associated membership variables

plike: array of shape(n_samples),
the likelihood under the prior

kfold: int, or array of shape(n_samples), optional,
folds in the cross-validation loop

Returns

like: array od shape(n_samples),
the (cross-validated) likelihood of the data

`estimate`(*x*, *niter=100*, *delta=0.0001*, *verbose=0*)

Estimation of the model given a dataset x

Parameters

x array of shape (n_samples,dim)
the data from which the model is estimated

niter=100: maximal number of iterations in the estimation process
delta = 1.e-4: increment of data likelihood at which
convergence is declared

verbose=0: verbosity mode

Returns

bic
[an asymptotic approximation of model evidence]

`evidence`(*x*, *z*, *nperm=0*, *verbose=0*)

See bayes_factor(self, x, z, nperm=0, verbose=0)

`guess_priors`(*x*, *nocheck=0*)

Set the priors in order of having them weakly uninformative this is from Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x, array of shape (nb_samples,self.dim)
the data used in the estimation process

nocheck: boolean, optional,
if nocheck==True, check is skipped

`guess_regularizing`(*x*, *bcheck=1*)

Set the regularizing priors as weakly informative according to Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x array of shape (n_samples,dim)
the data used in the estimation process

`initialize`(*x*)

initialize z using a k-means algorithm, then update the parameters

Parameters

x: array of shape (nb_samples,self.dim)
the data used in the estimation process

`initialize_and_estimate`(*x*, *z=None*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)

Estimation of self given x

Parameters

x array of shape (n_samples,dim)
the data from which the model is estimated

z = None: array of shape (n_samples)
a prior labelling of the data to initialize the computation

niter=100: maximal number of iterations in the estimation process
delta = 1.e-4: increment of data likelihood at which
convergence is declared

ninit=1: number of initialization performed
to reach a good solution

verbose=0: verbosity mode

Returns

the best model is returned

`likelihood`(*x*, *plike=None*)

return the likelihood of the model for the data x the values are weighted by the components weights

Parameters

x: array of shape (n_samples, self.dim),
the data used in the estimation process

plike: array of shape (n_samples), optional,
the density of each point under the prior

Returns

like, array of shape (nbitem, self.k)
component-wise likelihood

`likelihood_under_the_prior`(*x*)

Computes the likelihood of x under the prior

Parameters

x, array of shape (self.n_samples,self.dim)

Returns

w, the likelihood of x under the prior model (unweighted)

`map_label`(*x*, *like=None*)

return the MAP labelling of x

Parameters

x array of shape (n_samples,dim)
the data under study

like=None array of shape(n_samples,self.k)
component-wise likelihood if like==None, it is recomputed

Returns

z: array of shape(n_samples): the resulting MAP labelling
of the rows of x

`mixture_likelihood`(*x*)

Returns the likelihood of the mixture for x

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

`plugin`(*means*, *precisions*, *weights*)

Set manually the weights, means and precision of the model

Parameters

means: array of shape (self.k,self.dim)
precisions: array of shape (self.k,self.dim,self.dim)
or (self.k, self.dim)

weights: array of shape (self.k)

`pop`(*z*)

compute the population, i.e. the statistics of allocation

Parameters

z array of shape (nb_samples), type = np.int_
the allocation variable

Returns

hist
[array shape (self.k) count variable]

`probability_under_prior`()

Compute the probability of the current parameters of self given the priors

`reduce`(*z*)

Reduce the assignments by removing empty clusters and update self.k

Parameters

z: array of shape(n),
a vector of membership variables changed in place

Returns

z: the remapped values

`sample`(*x*, *niter=1*, *sampling_points=None*, *init=False*, *kfold=None*, *verbose=0*)

sample the indicator and parameters

Parameters

x: array of shape (n_samples, self.dim)
the data used in the estimation process

niter: int,
the number of iterations to perform

sampling_points: array of shape(nbpoints, self.dim), optional
points where the likelihood will be sampled this defaults to x

kfold: int or array, optional,
parameter of cross-validation control by default, no cross-validation is used the procedure is faster but less accurate

verbose=0: verbosity mode

Returns

likelihood: array of shape(nbpoints)
total likelihood of the model

`sample_and_average`(*x*, *niter=1*, *verbose=0*)

sample the indicator and parameters the average values for weights,means, precisions are returned

Parameters

x = array of shape (nb_samples,dim)
the data from which bic is computed

niter=1: number of iterations

Returns

weights: array of shape (self.k)
means: array of shape (self.k,self.dim)
precisions: array of shape (self.k,self.dim,self.dim)
or (self.k, self.dim) these are the average parameters across samplings

Notes

All this makes sense only if no label switching as occurred so this is wrong in general (asymptotically).

fix: implement a permutation procedure for components identification

`sample_indicator`(*like*)

Sample the indicator from the likelihood

Parameters

like: array of shape (nbitem,self.k)
component-wise likelihood

Returns

z: array of shape(nbitem): a draw of the membership variable

Notes

The behaviour is different from standard bgmm in that z can take arbitrary values

`set_constant_densities`(*prior_dens=None*)

Set the null and prior densities as constant (assuming a compact domain)

Parameters

prior_dens: float, optional
constant for the prior density

`set_priors`(*x*)

Set the priors in order of having them weakly uninformative this is from Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x, array of shape (n_samples,self.dim)
the data used in the estimation process

`show`(*x*, *gd*, *density=None*, *axes=None*)

Function to plot a GMM, still in progress Currently, works only in 1D and 2D

Parameters

x: array of shape(n_samples, dim)
the data under study

gd: GridDescriptor instance
density: array os shape(prod(gd.n_bins))
density of the model one the discrete grid implied by gd by default, this is recomputed

`show_components`(*x*, *gd*, *density=None*, *mpaxes=None*)

Function to plot a GMM – Currently, works only in 1D

Parameters

x: array of shape(n_samples, dim)
the data under study

gd: GridDescriptor instance
density: array os shape(prod(gd.n_bins))
density of the model one the discrete grid implied by gd by default, this is recomputed

mpaxes: axes handle to make the figure, optional,
if None, a new figure is created

`simple_update`(*x*, *z*, *plike*)

This is a step in the sampling procedure

that uses internal corss_validation

Parameters

x: array of shape(n_samples, dim),
the input data

z: array of shape(n_samples),
the associated membership variables

plike: array of shape(n_samples),
the likelihood under the prior

Returns

like: array od shape(n_samples),
the likelihood of the data

`test`(*x*, *tiny=1e-15*)

Returns the log-likelihood of the mixture for x

Parameters

x array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

ll: array of shape(n_samples)
the log-likelihood of the rows of x

`train`(*x*, *z=None*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)

Idem initialize_and_estimate

`unweighted_likelihood`(*x*)

return the likelihood of each data for each component the values are not weighted by the component weights

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
unweighted component-wise likelihood

Notes

Hopefully faster

`unweighted_likelihood_`(*x*)

return the likelihood of each data for each component the values are not weighted by the component weights

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
unweighted component-wise likelihood

`update`(*x*, *z*)

Update function (draw a sample of the IMM parameters)

Parameters

x array of shape (n_samples,self.dim)
the data used in the estimation process

z array of shape (n_samples), type = np.int_
the corresponding classification

`update_means`(*x*, *z*)

Given the allocation vector z, and the corresponding data x, resample the mean

Parameters

x: array of shape (nb_samples,self.dim)
the data used in the estimation process

z: array of shape (nb_samples), type = np.int_
the corresponding classification

`update_precisions`(*x*, *z*)

Given the allocation vector z, and the corresponding data x, resample the precisions

Parameters

x array of shape (nb_samples,self.dim)
the data used in the estimation process

z array of shape (nb_samples), type = np.int_
the corresponding classification

`update_weights`(*z*)

Given the allocation vector z, resmaple the weights parameter

Parameters

z array of shape (n_samples), type = np.int_
the allocation variable

24.2.2 `MixedIMM`

class `nipy.algorithms.clustering.imm.`**`MixedIMM`**(*alpha=0.5*, *dim=1*)

Bases: *`IMM`*

Particular IMM with an additional null class. The data is supplied together with a sample-related probability of being under the null.

`__init__`(*alpha=0.5*, *dim=1*)

Parameters

alpha: float, optional,
the parameter for cluster creation

dim: int, optional,
the dimension of the the data

Note: use the function set_priors() to set adapted priors

`average_log_like`(*x*, *tiny=1e-15*)

returns the averaged log-likelihood of the mode for the dataset x

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

tiny = 1.e-15: a small constant to avoid numerical singularities

`bayes_factor`(*x*, *z*, *nperm=0*, *verbose=0*)

Evaluate the Bayes Factor of the current model using Chib's method

Parameters

x: array of shape (nb_samples,dim)
the data from which bic is computed

z: array of shape (nb_samples), type = np.int_
the corresponding classification

nperm=0: int
the number of permutations to sample to model the label switching issue in the computation of the Bayes Factor By default, exhaustive permutations are used

verbose=0: verbosity mode

Returns

bf (float) the computed evidence (Bayes factor)

Notes

See: Marginal Likelihood from the Gibbs Output Journal article by Siddhartha Chib; Journal of the American Statistical Association, Vol. 90, 1995

`bic`(*like*, *tiny=1e-15*)

Computation of bic approximation of evidence

Parameters

like, array of shape (n_samples, self.k)
component-wise likelihood

tiny=1.e-15, a small constant to avoid numerical singularities

Returns

the bic value, float

`check`()

Checking the shape of sifferent matrices involved in the model

`check_x`(*x*)

essentially check that x.shape[1]==self.dim

x is returned with possibly reshaping

`conditional_posterior_proba`(*x*, *z*, *perm=None*)

Compute the probability of the current parameters of self given x and z

Parameters

x: array of shape (nb_samples, dim),
the data from which bic is computed

z: array of shape (nb_samples), type = np.int_,
the corresponding classification

perm: array ok shape(nperm, self.k),typ=np.int_, optional
all permutation of z under which things will be recomputed By default, no permutation is performed

`cross_validated_update`(*x*, *z*, *plike*, *null_class_proba*, *kfold=10*)

This is a step in the sampling procedure that uses internal corss_validation

Parameters

x: array of shape(n_samples, dim),
the input data

z: array of shape(n_samples),
the associated membership variables

plike: array of shape(n_samples),
the likelihood under the prior

kfold: int, optional, or array
number of folds in cross-validation loop or set of indexes for the cross-validation procedure

null_class_proba: array of shape(n_samples),
prior probability to be under the null

Returns

like: array od shape(n_samples),
the (cross-validated) likelihood of the data

z: array of shape(n_samples),
the associated membership variables

Notes

When kfold is an array, there is an internal reshuffling to randomize the order of updates

`estimate`(*x*, *niter=100*, *delta=0.0001*, *verbose=0*)

Estimation of the model given a dataset x

Parameters

x array of shape (n_samples,dim)
the data from which the model is estimated

niter=100: maximal number of iterations in the estimation process
delta = 1.e-4: increment of data likelihood at which
convergence is declared

verbose=0: verbosity mode

Returns

bic
[an asymptotic approximation of model evidence]

`evidence`(*x*, *z*, *nperm=0*, *verbose=0*)

See bayes_factor(self, x, z, nperm=0, verbose=0)

`guess_priors`(*x*, *nocheck=0*)

Set the priors in order of having them weakly uninformative this is from Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x, array of shape (nb_samples,self.dim)
the data used in the estimation process

nocheck: boolean, optional,
if nocheck==True, check is skipped

`guess_regularizing`(*x*, *bcheck=1*)

Set the regularizing priors as weakly informative according to Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x array of shape (n_samples,dim)
the data used in the estimation process

`initialize`(*x*)

initialize z using a k-means algorithm, then update the parameters

Parameters

x: array of shape (nb_samples,self.dim)
the data used in the estimation process

`initialize_and_estimate`(*x*, *z=None*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)

Estimation of self given x

Parameters

x array of shape (n_samples,dim)
the data from which the model is estimated

z = None: array of shape (n_samples)
a prior labelling of the data to initialize the computation

niter=100: maximal number of iterations in the estimation process
delta = 1.e-4: increment of data likelihood at which
convergence is declared

ninit=1: number of initialization performed
to reach a good solution

verbose=0: verbosity mode

Returns

the best model is returned

`likelihood`(*x*, *plike=None*)

return the likelihood of the model for the data x the values are weighted by the components weights

Parameters

x: array of shape (n_samples, self.dim),
the data used in the estimation process

plike: array of shape (n_samples), optional,
the density of each point under the prior

Returns

like, array of shape (nbitem, self.k)
component-wise likelihood

`likelihood_under_the_prior`(*x*)

Computes the likelihood of x under the prior

Parameters

x, array of shape (self.n_samples,self.dim)

Returns

w, the likelihood of x under the prior model (unweighted)

`map_label`(*x*, *like=None*)

return the MAP labelling of x

Parameters

x array of shape (n_samples,dim)
the data under study

like=None array of shape(n_samples,self.k)
component-wise likelihood if like==None, it is recomputed

Returns

z: array of shape(n_samples): the resulting MAP labelling
of the rows of x

`mixture_likelihood`(*x*)

Returns the likelihood of the mixture for x

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

`plugin`(*means*, *precisions*, *weights*)

Set manually the weights, means and precision of the model

Parameters

means: array of shape (self.k,self.dim)
precisions: array of shape (self.k,self.dim,self.dim)
or (self.k, self.dim)

weights: array of shape (self.k)

`pop`(*z*)

compute the population, i.e. the statistics of allocation

Parameters

z array of shape (nb_samples), type = np.int_
the allocation variable

Returns

hist
[array shape (self.k) count variable]

`probability_under_prior`()

Compute the probability of the current parameters of self given the priors

`reduce`(*z*)

Reduce the assignments by removing empty clusters and update self.k

Parameters

z: array of shape(n),
a vector of membership variables changed in place

Returns

z: the remapped values

`sample`(*x*, *null_class_proba*, *niter=1*, *sampling_points=None*, *init=False*, *kfold=None*, *co_clustering=False*, *verbose=0*)

sample the indicator and parameters

Parameters

x: array of shape (n_samples, self.dim),
the data used in the estimation process

null_class_proba: array of shape(n_samples),
the probability to be under the null

niter: int,
the number of iterations to perform

sampling_points: array of shape(nbpoints, self.dim), optional
points where the likelihood will be sampled this defaults to x

kfold: int, optional,
parameter of cross-validation control by default, no cross-validation is used the procedure is faster but less accurate

co_clustering: bool, optional
if True, return a model of data co-labelling across iterations

verbose=0: verbosity mode

Returns

likelihood: array of shape(nbpoints)
total likelihood of the model

pproba: array of shape(n_samples),
the posterior of being in the null (the posterior of null_class_proba)

coclust: only if co_clustering==True,
sparse_matrix of shape (n_samples, n_samples), frequency of co-labelling of each sample pairs across iterations

`sample_and_average`(*x*, *niter=1*, *verbose=0*)

sample the indicator and parameters the average values for weights,means, precisions are returned

Parameters

x = array of shape (nb_samples,dim)
the data from which bic is computed

niter=1: number of iterations

Returns

weights: array of shape (self.k)
means: array of shape (self.k,self.dim)
precisions: array of shape (self.k,self.dim,self.dim)
or (self.k, self.dim) these are the average parameters across samplings

Notes

All this makes sense only if no label switching as occurred so this is wrong in general (asymptotically).

fix: implement a permutation procedure for components identification

`sample_indicator`(*like*, *null_class_proba*)

sample the indicator from the likelihood

Parameters

like: array of shape (nbitem,self.k)
component-wise likelihood

null_class_proba: array of shape(n_samples),
prior probability to be under the null

Returns

z: array of shape(nbitem): a draw of the membership variable

Notes

Here z=-1 encodes for the null class

`set_constant_densities`(*null_dens=None*, *prior_dens=None*)

Set the null and prior densities as constant (over a supposedly compact domain)

Parameters

null_dens: float, optional
constant for the null density

prior_dens: float, optional
constant for the prior density

`set_priors`(*x*)

Set the priors in order of having them weakly uninformative this is from Fraley and raftery; Journal of Classification 24:155-181 (2007)

Parameters

x, array of shape (n_samples,self.dim)
the data used in the estimation process

`show`(*x*, *gd*, *density=None*, *axes=None*)

Function to plot a GMM, still in progress Currently, works only in 1D and 2D

Parameters

x: array of shape(n_samples, dim)
the data under study

gd: GridDescriptor instance
density: array os shape(prod(gd.n_bins))
density of the model one the discrete grid implied by gd by default, this is recomputed

`show_components`(*x*, *gd*, *density=None*, *mpaxes=None*)

Function to plot a GMM – Currently, works only in 1D

Parameters

x: array of shape(n_samples, dim)
the data under study

gd: GridDescriptor instance
density: array os shape(prod(gd.n_bins))
density of the model one the discrete grid implied by gd by default, this is recomputed

mpaxes: axes handle to make the figure, optional,
if None, a new figure is created

`simple_update`(*x*, *z*, *plike*, *null_class_proba*)
One step in the sampling procedure (one data sweep)

Parameters

x: array of shape(n_samples, dim),
the input data

z: array of shape(n_samples),
the associated membership variables

plike: array of shape(n_samples),
the likelihood under the prior

null_class_proba: array of shape(n_samples),
prior probability to be under the null

Returns

like: array od shape(n_samples),
the likelihood of the data under the H1 hypothesis

`test`(*x*, *tiny=1e-15*)
Returns the log-likelihood of the mixture for x

Parameters

x array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

ll: array of shape(n_samples)
the log-likelihood of the rows of x

`train`(*x*, *z=None*, *niter=100*, *delta=0.0001*, *ninit=1*, *verbose=0*)
Idem initialize_and_estimate

`unweighted_likelihood`(*x*)
return the likelihood of each data for each component the values are not weighted by the component weights

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
unweighted component-wise likelihood

Notes

Hopefully faster

`unweighted_likelihood_`(*x*)

return the likelihood of each data for each component the values are not weighted by the component weights

Parameters

x: array of shape (n_samples,self.dim)
the data used in the estimation process

Returns

like, array of shape(n_samples,self.k)
unweighted component-wise likelihood

`update`(*x*, *z*)

Update function (draw a sample of the IMM parameters)

Parameters

x array of shape (n_samples,self.dim)
the data used in the estimation process

z array of shape (n_samples), type = np.int_
the corresponding classification

`update_means`(*x*, *z*)

Given the allocation vector z, and the corresponding data x, resample the mean

Parameters

x: array of shape (nb_samples,self.dim)
the data used in the estimation process

z: array of shape (nb_samples), type = np.int_
the corresponding classification

`update_precisions`(*x*, *z*)

Given the allocation vector z, and the corresponding data x, resample the precisions

Parameters

x array of shape (nb_samples,self.dim)
the data used in the estimation process

z array of shape (nb_samples), type = np.int_
the corresponding classification

`update_weights`(*z*)

Given the allocation vector z, resmaple the weights parameter

Parameters

z array of shape (n_samples), type = np.int_
the allocation variable

24.3 Functions

`nipy.algorithms.clustering.imm.`**`co_labelling`**(*z*, *kmax=None*, *kmin=None*)

return a sparse co-labelling matrix given the label vector z

Parameters

z: array of shape(n_samples),
the input labels

kmax: int, optional,
considers only the labels in the range [0, kmax[

Returns

colabel: a sparse coo_matrix,
yields the co labelling of the data i.e. c[i,j]= 1 if z[i]==z[j], 0 otherwise

`nipy.algorithms.clustering.imm.`**`main`**()

Illustrative example of the behaviour of imm

CHAPTER

TWENTYFIVE

ALGORITHMS.CLUSTERING.UTILS

25.1 Module: `algorithms.clustering.utils`

25.2 Functions

`nipy.algorithms.clustering.utils.`**`kmeans`**(*X*, *nbclusters=2*, *Labels=None*, *maxiter=300*, *delta=0.0001*, *verbose=0*, *ninit=1*)

kmeans clustering algorithm

Parameters

X: array of shape (n,p): n = number of items, p = dimension
data array

nbclusters (int), the number of desired clusters
Labels = None array of shape (n) prior Labels.
if None or inadequate a random initialization is performed.

maxiter=300 (int), the maximum number of iterations before convergence
delta: float, optional,
the relative increment in the results before declaring convergence.

verbose: verbosity mode, optional
ninit: int, optional, number of random initializations

Returns

Centers: array of shape (nbclusters, p),
the centroids of the resulting clusters

Labels
[array of size n, the discrete labels of the input items]

J (float): the final value of the inertia criterion

`nipy.algorithms.clustering.utils.`**`voronoi`**(*x*, *centers*)

Assignment of data items to nearest cluster center

Parameters

x array of shape (n,p)
n = number of items, p = data dimension

centers, array of shape (k, p) the cluster centers

Returns

z vector of shape(n), the resulting assignment

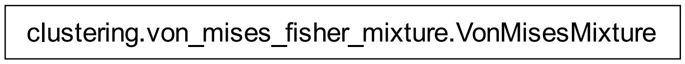
CHAPTER

TWENTYSIX

ALGORITHMS.CLUSTERING.VON_MISES_FISHER_MIXTURE

26.1 Module: `algorithms.clustering.von_mises_fisher_mixture`

Inheritance diagram for `nipy.algorithms.clustering.von_mises_fisher_mixture`:



Implementation of Von-Mises-Fisher Mixture models, i.e. the equivalent of mixture of Gaussian on the sphere.

Author: Bertrand Thirion, 2010-2011

26.2 Class

26.3 `VonMisesMixture`

class `nipy.algorithms.clustering.von_mises_fisher_mixture.`**`VonMisesMixture`**(*k*, *precision*, *means=None*, *weights=None*, *null_class=False*)

Bases: `object`

Model for Von Mises mixture distribution with fixed variance on a two-dimensional sphere

`__init__`(*k*, *precision*, *means=None*, *weights=None*, *null_class=False*)

Initialize Von Mises mixture

Parameters

k: int,
number of components

precision: float,
the fixed precision parameter

means: array of shape(self.k, 3), optional
input component centers

weights: array of shape(self.k), optional
input components weights

null_class: bool, optional
Inclusion of a null class within the model (related to k=0)

`density_per_component`(*x*)

Compute the per-component density of the data

Parameters

x: array of shape(n,3)
should be on the unit sphere

Returns

like: array of shape(n, self.k), with non-neagtive values
the density

`estimate`(*x*, *maxiter=100*, *miniter=1*, *bias=None*)

Return average log density across samples

Parameters

x: array of shape (n,3)
should be on the unit sphere

maxiter
[int, optional] maximum number of iterations of the algorithms

miniter
[int, optional] minimum number of iterations

bias
[array of shape(n), optional] prior probability of being in a non-null class

Returns

ll
[float] average (across samples) log-density

`estimate_means`(*x*, *z*)

Calculate and set means from *x* and *z*

Parameters

x: array of shape(n,3)
should be on the unit sphere

z: array of shape(self.k)

`estimate_weights`(*z*)

Calculate and set weights from *z*

Parameters

z: array of shape(self.k)

`log_density_per_component`(*x*)

Compute the per-component density of the data

Parameters

x: array of shape(n,3)
should be on the unit sphere

Returns

like: array of shape(n, self.k), with non-neagtive values
the density

`log_weighted_density`(*x*)
Return log weighted density

Parameters

x: array of shape(n,3)
should be on the unit sphere

Returns

log_like: array of shape(n, self.k)

`mixture_density`(*x*)
Return mixture density

Parameters

x: array of shape(n,3)
should be on the unit sphere

Returns

like: array of shape(n)

`responsibilities`(*x*)
Return responsibilities

Parameters

x: array of shape(n,3)
should be on the unit sphere

Returns

resp: array of shape(n, self.k)

`show`(*x*)
Visualization utility

Parameters

x: array of shape (n, 3)
should be on the unit sphere

Notes

Uses `matplotlib`.

`weighted_density`(*x*)
Return weighted density

Parameters

x: array shape(n,3)
should be on the unit sphere

Returns

like: array
of shape(n, self.k)

26.4 Functions

`nipy.algorithms.clustering.von_mises_fisher_mixture.`**`estimate_robust_vmm`**(*k*, *precision*, *null_class*, *x*, *ninit=10*, *bias=None*, *maxiter=100*)

Return the best von_mises mixture after severla initialization

Parameters

k: int, number of classes
precision: float, priori precision parameter
null class: bool, optional,
should a null class be included or not

x: array of shape(n,3)
input data, should be on the unit sphere

ninit: int, optional,
number of iterations

bias: array of shape(n), optional
prior probability of being in a non-null class

maxiter: int, optional,
maximum number of iterations after each initialization

`nipy.algorithms.clustering.von_mises_fisher_mixture.`**`example_cv_nonoise`**()

`nipy.algorithms.clustering.von_mises_fisher_mixture.`**`example_noisy`**()

`nipy.algorithms.clustering.von_mises_fisher_mixture.`**`select_vmm`**(*krange*, *precision*, *null_class*, *x*, *ninit=10*, *bias=None*, *maxiter=100*, *verbose=0*)

Return the best von_mises mixture after severla initialization

Parameters

krange: list of ints,
number of classes to consider

precision:
null class:
x: array of shape(n,3)
should be on the unit sphere

ninit: int, optional,
number of iterations

maxiter: int, optional,
bias: array of shape(n),
a prior probability of not being in the null class

verbose: Bool, optional

nipy.algorithms.clustering.von_mises_fisher_mixture.**select_vmm_cv**(*krange*, *precision*, *x*, *null_class*, *cv_index*, *ninit=5*, *maxiter=100*, *bias=None*, *verbose=0*)

Return the best von_mises mixture after severla initialization

Parameters

krange: list of ints,
number of classes to consider

precision: float,
precision parameter of the von-mises densities

x: array of shape(n, 3)
should be on the unit sphere

null class: bool, whether a null class should be included or not
cv_index: set of indices for cross validation
ninit: int, optional,
number of iterations

maxiter: int, optional,
bias: array of shape (n), prior

nipy.algorithms.clustering.von_mises_fisher_mixture.**sphere_density**(*npoints*)

Return the points and area of a npoints**2 points sampled on a sphere

Returns

s
[array of shape(npoints ** 2, 3)]

area: array of shape(npoints)

CHAPTER

TWENTYSEVEN

ALGORITHMS.DIAGNOSTICS.COMMANDS

27.1 Module: `algorithms.diagnostics.commands`

Implementation of diagnostic command line tools

Tools are:

- nipy_diagnose
- nipy_tsdiffana

This module has the logic for each command.

The command script files deal with argument parsing and any custom imports. The implementation here accepts the `args` object from `argparse` and does the work.

27.2 Functions

`nipy.algorithms.diagnostics.commands.`**`diagnose`**`(`*args*`)`

Calculate, write results from diagnostic screen

Parameters

args
[object] object with attributes:

- filename : str - 4D image filename
- time_axis : str - name or number of time axis in *filename*
- slice_axis : str - name or number of slice axis in *filename*
- out_path : None or str - path to which to write results
- out_fname_label : None or filename - suffix of output results files
- ncomponents : int - number of PCA components to write images for

Returns

res
[dict] Results of running `screen()` on *filename*

`nipy.algorithms.diagnostics.commands.`**`parse_fname_axes`**`(`*img_fname*, *time_axis*, *slice_axis*`)`

Load *img_fname*, check *time_axis*, *slice_axis* or use default

Parameters

img_fname
[str] filename of image on which to do diagnostics

time_axis
[None or str or int, optional] Axis indexing time-points. None is default, will be replaced by a value of 't'. If *time_axis* is an integer, gives the index of the input (domain) axis of *img*. If *time_axis* is a str, can be an input (domain) name, or an output (range) name, that maps to an input (domain) name.

slice_axis
[None or str or int, optional] Axis indexing MRI slices. If *slice_axis* is an integer, gives the index of the input (domain) axis of *img*. If *slice_axis* is a str, can be an input (domain) name, or an output (range) name, that maps to an input (domain) name. If None (the default) then 1) try the name 'slice' to select the axis - if this fails, and *fname* refers to an Analyze type image (such as Nifti), then 2) default to the third image axis, otherwise 3) raise a ValueError

Returns

img
[`Image` instance] Image as loaded from *img_fname*

time_axis
[int or str] Time axis, possibly filled with default

slice_axis
[int or str] Slice axis, possibly filled with default

`nipy.algorithms.diagnostics.commands.`**`tsdiffana`**(*args*)

Generate tsdiffana plots from command line params *args*

Parameters

args
[object] object with attributes

- filename : str - 4D image filename
- out_file : str - graphics file to write to instead of leaving graphics on screen
- time_axis : str - name or number of time axis in *filename*
- slice_axis : str - name or number of slice axis in *filename*
- write_results : bool - if True, write images and plots to files
- out_path : None or str - path to which to write results
- out_fname_label : None or filename - suffix of output results files

Returns

axes
[Matplotlib axes] Axes on which we have done the plots.

CHAPTER

TWENTYEIGHT

ALGORITHMS.DIAGNOSTICS.SCREENS

28.1 Module: `algorithms.diagnostics.screens`

Diagnostic 4d image screen

28.2 Functions

`nipy.algorithms.diagnostics.screens.`**`screen`**(*img4d*, *ncomp=10*, *time_axis='t'*, *slice_axis=None*)

Diagnostic screen for 4d FMRI image

Includes PCA, tsdiffana and mean, std, min, max images.

Parameters

img4d
[`Image`] 4d image file

ncomp
[int, optional] number of component images to return. Default is 10

time_axis
[str or int, optional] Axis over which to do PCA, time difference analysis. Defaults to *t*

slice_axis
[None or str or int, optional] Name or index of input axis over which to do slice analysis for time difference analysis. If None, look for input axis `slice`. At the moment we then assume slice is the last non-time axis, but this last guess we will remove in future versions of nipy. The default will then be 'slice' and you'll get an error if there is no axis named 'slice'.

Returns

screen
[dict] with keys:

- mean : mean image (all summaries are over last dimension)
- std : standard deviation image
- max : image of max
- min : min
- pca : 4D image of PCA component images
- pca_res : dict of results from PCA
- ts_res : dict of results from tsdiffana

Examples

```
>>> import nipy as ni
>>> from nipy.testing import funcfile
>>> img = ni.load_image(funcfile)
>>> screen_res = screen(img)
>>> screen_res['mean'].ndim
3
>>> screen_res['pca'].ndim
4
```

`nipy.algorithms.diagnostics.screens.`**`write_screen_res`**(*res*, *out_path*, *out_root*, *out_img_ext='.nii'*, *pcnt_var_thresh=0.1*)

Write results from `screen` to disk as images

Parameters

res
[dict] output from `screen` function

out_path
[str] directory to which to write output images

out_root
[str] part of filename between image-specific prefix and image-specific extension to use for writing images

out_img_ext
[str, optional] extension (identifying image type) to which to write volume images. Default is '.nii'

pcnt_var_thresh
[float, optional] threshold below which we do not plot percent variance explained by components; default is 0.1. This removes the long tail from percent variance plots.

Returns

None

CHAPTER

TWENTYNINE

ALGORITHMS.DIAGNOSTICS.TIMEDIFF

29.1 Module: `algorithms.diagnostics.timediff`

Time series diagnostics

These started life as `tsdiffana.m` - see http://imaging.mrc-cbu.cam.ac.uk/imaging/DataDiagnostics

Oliver Josephs (FIL) gave me (MB) the idea of time-point to time-point subtraction as a diagnostic for motion and other sudden image changes.

29.2 Functions

`nipy.algorithms.diagnostics.timediff.`**`time_slice_diffs`**(*arr*, *time_axis=-1*, *slice_axis=None*)

Time-point to time-point differences over volumes and slices

We think of the passed array as an image. The image has a "time" dimension given by *time_axis* and a "slice" dimension, given by *slice_axis*, and one or more other dimensions. In the case of imaging there will usually be two more dimensions (the dimensions defining the size of an image slice). A single slice in the time dimension we call a "volume". A single entry in *arr* is a "voxel". For example, if *time_axis* == 0, then `v = arr[0]` would be the first volume in the series. The volume `v` above has `v.size` voxels. If, in addition, *slice_axis* == 1, then for the volume `v` (above) `s = v[0]` would be a "slice", with `s.size` voxels. These are obviously terms from neuroimaging.

Parameters

arr
[array_like] Array over which to calculate time and slice differences. We'll call this array an 'image' in this doc.

time_axis
[int, optional] axis of *arr* that varies over time. Default is last

slice_axis
[None or int, optional] axis of *arr* that varies over image slice. None gives last non-time axis.

Returns

results
[dict] `T` is the number of time points (`arr.shape[time_axis]`)

`S` is the number of slices (`arr.shape[slice_axis]`)

`v` is the shape of a volume (`rollimg(arr, time_axis)[0].shape`) `d2[t]` is the volume of squared differences between voxels at time point `t` and time point `t+1`

results has keys:

- **'volume_mean_diff2'**
  [(T-1,) array] array containing the mean (over voxels in volume) of the squared difference from one time point to the next
- **'slice_mean_diff2'**
  [(T-1, S) array] giving the mean (over voxels in slice) of the difference from one time point to the next, one value per slice, per timepoint
- **'volume_means'**
  [(T,) array] mean over voxels for each volume `vol[t] for t in 0:T`
- **'slice_diff2_max_vol'**
  [v[:] array] volume, of same shape as input time point volumes, where each slice is is the slice from `d2[t]` for t in 0:T-1, that has the largest variance across `t`. Thus each slice in the volume may well result from a different difference time point.
- **'diff2_mean_vol``**
  [v[:] array] volume with the mean of `d2[t]` across t for t in 0:T-1.

Raises

ValueError
[if *time_axis* refers to same axis as *slice_axis*]

`nipy.algorithms.diagnostics.timediff.`**`time_slice_diffs_image`**`(`*img*, *time_axis='t'*, *slice_axis='slice'*`)`

Time-point to time-point differences over volumes and slices of image

Parameters

img
[Image] The image on which to perform time-point differences

time_axis
[str or int, optional] Axis indexing time-points. Default is 't'. If *time_axis* is an integer, gives the index of the input (domain) axis of *img*. If *time_axis* is a str, can be an input (domain) name, or an output (range) name, that maps to an input (domain) name.

slice_axis
[str or int, optional] Axis indexing MRI slices. If *slice_axis* is an integer, gives the index of the input (domain) axis of *img*. If *slice_axis* is a str, can be an input (domain) name, or an output (range) name, that maps to an input (domain) name.

Returns

results
[dict] *arr* refers to the array as loaded from *img*

`T` is the number of time points (`img.shape[time_axis]`)

`S` is the number of slices (`img.shape[slice_axis]`)

`v` is the shape of a volume (`rollimg(img, time_axis)[0].shape`)

`d2[t]` is the volume of squared differences between voxels at time point `t` and time point `t+1`

results has keys:

- **'volume_mean_diff2'**
  [(T-1,) array] array containing the mean (over voxels in volume) of the squared difference from one time point to the next
- **'slice_mean_diff2'**
  [(T-1, S) array] giving the mean (over voxels in slice) of the difference from one time point to the next, one value per slice, per timepoint
- **'volume_means'**
  [(T,) array] mean over voxels for each volume `vol[t] for t in 0:T`
- **'slice_diff2_max_vol'**
  [v[:] image] image volume, of same shape as input time point volumes, where each slice is is the slice from `d2[t]` for t in 0:T-1, that has the largest variance across `t`. Thus each slice in the volume may well result from a different difference time point.
- **'diff2_mean_vol``**
  [v[:] image] image volume with the mean of `d2[t]` across t for t in 0:T-1.

CHAPTER

THIRTY

ALGORITHMS.DIAGNOSTICS.TSDIFFPLOT

30.1 Module: `algorithms.diagnostics.tsdiffplot`

plot tsdiffana parameters

30.2 Functions

`nipy.algorithms.diagnostics.tsdiffplot.`**`plot_tsdiffs`**`(`*results*, *axes=None*`)`

Plotting routine for time series difference metrics

Requires matplotlib

Parameters

results
[dict] Results of format returned from `nipy.algorithms.diagnostics.time_slice_diff()`

`nipy.algorithms.diagnostics.tsdiffplot.`**`plot_tsdiffs_image`**`(`*img*, *axes=None*, *show=True*`)`

plot_tsdiffs_image is deprecated! Please see docstring for alternative code

Plot time series diagnostics for image

This function is deprecated; please use something like:

```
results = time_slice_diff_image(img, slice_axis=2)
plot_tsdiffs(results)
```

instead.

Parameters

img
[image-like or filename str] image on which to do diagnostics

axes
[None or sequence, optional] Axes on which to plot the diagnostics. If None, then we create a figure and subplots for the plots. Sequence should have length >=4.

show
[{True, False}, optional] If True, show the figure after plotting it

Returns

axes
[Matplotlib axes] Axes on which we have done the plots. Will be same as *axes* input if *axes* input was not None

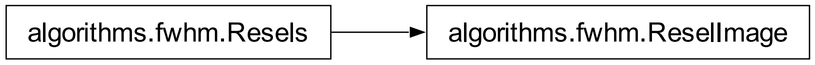
CHAPTER

THIRTYONE

ALGORITHMS.FWHM

31.1 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

31.2 Classes

31.2.1 `ReselImage`

class `nipy.algorithms.fwhm.`**`ReselImage`**(*resels=None*, *fwhm=None*, ***keywords*)

Bases: `Resels`

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

Initialize resel image

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*

`fwhm2resel`(*fwhm*)
Convert FWHM *fwhm* to equivalent reseels per voxel

Parameters

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

Returns

resels
[float]

`integrate`(*mask=None*)
Integrate resels within *mask* (or use self.mask)

Parameters

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

Returns

total_resels
the resels contained in the mask

FWHM
[float] an estimate of FWHM based on the average resel per voxel

nvoxel: int
the number of voxels in the mask

`resel2fwhm`(*resels*)
Convert resels as *resels* to isotropic FWHM

Parameters

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

Returns

fwhm
[float]

31.2.2 Resels

class `nipy.algorithms.fwhm.`**`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*)
Initialize resels class

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*)

Convert FWHM *fwhm* to equivalent reseels per voxel

Parameters

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

Returns

resels
[float]

`integrate`(*mask=None*)

Integrate resels within *mask* (or use self.mask)

Parameters

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

Returns

total_resels
the resels contained in the mask

FWHM
[float] an estimate of FWHM based on the average resel per voxel

nvoxel: int
the number of voxels in the mask

`resel2fwhm`(*resels*)

Convert resels as *resels* to isotropic FWHM

Parameters

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

Returns

fwhm
[float]

CHAPTER

THIRTYTWO

ALGORITHMS.GRAPH.BIPARTITE_GRAPH

32.1 Module: `algorithms.graph.bipartite_graph`

Inheritance diagram for `nipy.algorithms.graph.bipartite_graph`:

This module implements the BipartiteGraph class, used to represent weighted bipartite graph: it contains two types of vertices, say 'left' and 'right'; then edges can only exist between 'left' and 'right' vertices. For simplicity the vertices of either side are labeled [1..V] and [1..W] respectively.

Author: Bertrand Thirion, 2006–2011

32.2 Class

32.3 `BipartiteGraph`

class `nipy.algorithms.graph.bipartite_graph.`**`BipartiteGraph`**(*V*, *W*, *edges=None*, *weights=None*)

Bases: `object`

Bipartite graph class

A graph for which there are two types of nodes, such that edges can exist only between nodes of type 1 and type 2 (not within) fields of this class: 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*)

Constructor

Parameters

V (int), the number of vertices of subset 1
W (int), the number of vertices of subset 2
edges=None: array of shape (self.E, 2)
the edge array of the graph

weights=None: array of shape (self.E)
the associated weights array

`copy()`

returns a copy of self

`set_edges`(*edges*)

Set edges to graph

sets self.edges=edges if

1. edges has a correct size
2. edges take values in [0..V-1]*[0..W-1]

Parameters

edges: array of shape(self.E, 2): set of candidate edges

`set_weights`(*weights*)

Set weights *weights* to edges

Parameters

weights, array of shape(self.V): edges weights

`subgraph_left`(*valid*, *renumb=True*)

Extraction of a subgraph

Parameters

valid, boolean array of shape self.V
renumb, boolean: renumbering of the (left) edges

Returns

G
[None or `BipartiteGraph` instance] A new BipartiteGraph instance with only the left vertices that are True. If sum(valid)==0, None is returned

`subgraph_right`(*valid*, *renumb=True*)

Extraction of a subgraph

Parameters

valid
[bool array of shape self.V]

renumb
[bool, optional] renumbering of the (right) edges

Returns

G
[None or `BipartiteGraph` instance.] A new BipartiteGraph instance with only the right vertices that are True. If sum(valid)==0, None is returned

32.4 Functions

`nipy.algorithms.graph.bipartite_graph.`**`bipartite_graph_from_adjacency`**(*x*)

Instantiates a weighted graph from a square 2D array

Parameters

x: 2D array instance, the input array

Returns

wg: BipartiteGraph instance

`nipy.algorithms.graph.bipartite_graph.`**`bipartite_graph_from_coo_matrix`**(*x*)

Instantiates a weighted graph from a (sparse) coo_matrix

Parameters

x: scipy.sparse.coo_matrix instance, the input matrix

Returns

bg: BipartiteGraph instance

`nipy.algorithms.graph.bipartite_graph.`**`check_feature_matrices`**(*X*, *Y*)

checks whether the dimensions of X and Y are consistent

Parameters

X, Y arrays of shape (n1, p) and (n2, p)
where p = common dimension of the features

`nipy.algorithms.graph.bipartite_graph.`**`cross_eps`**(*X*, *Y*, *eps=1.0*)

Return the eps-neighbours graph of from X to Y

Parameters

X, Y arrays of shape (n1, p) and (n2, p)
where p = common dimension of the features
eps=1, float: the neighbourhood size considered

Returns

the resulting bipartite graph instance

Notes

for the sake of speed it is advisable to give PCA-preprocessed matrices X and Y.

`nipy.algorithms.graph.bipartite_graph.`**`cross_knn`**(*X*, *Y*, *k=1*)

return the k-nearest-neighbours graph of from X to Y

Parameters

X, Y arrays of shape (n1, p) and (n2, p)
where p = common dimension of the features
eps=1, float: the neighbourhood size considered

Returns

BipartiteGraph instance

Notes

For the sake of speed it is advised to give PCA-transformed matrices X and Y.

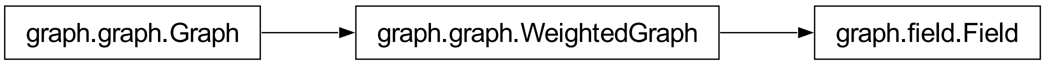
CHAPTER

THIRTYTHREE

ALGORITHMS.GRAPH.FIELD

33.1 Module: `algorithms.graph.field`

Inheritance diagram for `nipy.algorithms.graph.field`:



This module implements the Field class, which simply a WeightedGraph (see the graph.py) module, plus an array that yields (possibly multi-dimnesional) features associated with graph vertices. This allows some kinds of computations (all those relating to mathematical morphology, diffusion etc.)

Certain functions are provided to Instantiate Fields easily, given a WeightedGraph and feature data.

Author:Bertrand Thirion, 2006–2011

33.2 Class

33.3 `Field`

class `nipy.algorithms.graph.field.`**`Field`**(*V*, *edges=None*, *weights=None*, *field=None*)

Bases: *`WeightedGraph`*

This is the basic field structure,
which contains the weighted graph structure plus an array 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*)

Parameters

V (int > 0) the number of vertices of the graph
edges=None: the edge array of the graph
weights=None: the associated weights array
field=None: the field data itself

adjacency()

returns the adjacency matrix of the graph as a sparse coo matrix

Returns

adj: scipy.sparse matrix instance,
that encodes the adjacency matrix of self

anti_symmeterize()

anti-symmeterize self, i.e. produces the graph whose adjacency matrix would be the antisymmetric part of its current adjacency matrix

cc()

Compte the different connected components of the graph.

Returns

label: array of shape(self.V), labelling of the vertices

cliques()

Extraction of the graphe cliques these are defined using replicator dynamics equations

Returns

cliques: array of shape (self.V), type (np.int_)
labelling of the vertices according to the clique they belong to

closing(*nbiter=1*)

Morphological closing of the field data. self.field is changed inplace

Parameters

nbiter=1
[the number of iterations required]

compact_neighb()

returns a compact representation of self

Returns

idx: array of of shape(self.V + 1):
the positions where to find the neighbors of each node within neighb and weights

neighb: array of shape(self.E), concatenated list of neighbors
weights: array of shape(self.E), concatenated list of weights

constrained_voronoi(*seed*)

Voronoi parcellation of the field starting from the input seed

Parameters

seed: int array of shape(p), the input seeds

Returns

label: The resulting labelling of the data

Notes

FIXME: deal with graphs with several ccs

`copy()`

copy function

`custom_watershed(`*refdim=0*, *th=-inf*`)`

customized watershed analysis of the field. Note that bassins are found around each maximum (and not minimum as conventionally)

Parameters

refdim: int, optional
th: float optional, threshold of the field

Returns

idx: array of shape (nbassins)
indices of the vertices that are local maxima

label
[array of shape (self.V)] labelling of the vertices according to their bassin

`cut_redundancies()`

Returns a graph with redundant edges removed: ecah edge (ab) is present only once in the edge matrix: the correspondng weights are added.

Returns

the resulting WeightedGraph

`degrees()`

Returns the degree of the graph vertices.

Returns

rdegree: (array, type=int, shape=(self.V,)), the right degrees
ldegree: (array, type=int, shape=(self.V,)), the left degrees

`diffusion(`*nbiter=1*`)`

diffusion of the field data in the weighted graph structure self.field is changed inplace

Parameters

nbiter: int, optional the number of iterations required

Notes

The process is run for all the dimensions of the field

`dijkstra(`*seed=0*`)`

Returns all the [graph] geodesic distances starting from seed x

seed (int, >-1, <self.V) or array of shape(p)
edge(s) from which the distances are computed

Returns

dg: array of shape (self.V),
the graph distance dg from ant vertex to the nearest seed

Notes

It is mandatory that the graph weights are non-negative

`dilation`(*nbiter=1*, *fast=True*)

Morphological dilation of the field data, changed in place

Parameters

nbiter: int, optional, the number of iterations required

Notes

When data dtype is not float64, a slow version of the code is used

`erosion`(*nbiter=1*)

Morphological opening of the field

Parameters

nbiter: int, optional, the number of iterations required

`floyd`(*seed=None*)

Compute all the geodesic distances starting from seeds

Parameters

seed= None: array of shape (nbseed), type np.int_
vertex indexes from which the distances are computed if seed==None, then every edge is a seed point

Returns

dg array of shape (nbseed, self.V)
the graph distance dg from each seed to any vertex

Notes

It is mandatory that the graph weights are non-negative. The algorithm proceeds by repeating Dijkstra's algo for each seed. Floyd's algo is not used (O(self.V)^3 complexity. . .)

`from_3d_grid`(*xyz*, *k=18*)

Sets the graph to be the topological neighbours graph of the three-dimensional coordinates set xyz, in the k-connectivity scheme

Parameters

xyz: array of shape (self.V, 3) and type np.int_,
k = 18: the number of neighbours considered. (6, 18 or 26)

Returns

E(int): the number of edges of self

`geodesic_kmeans`(*seeds=None*, *label=None*, *maxiter=100*, *eps=0.0001*, *verbose=0*)

Geodesic k-means algorithm i.e. obtention of clusters that are topologically connected and minimally variable concerning the information of self.field

Parameters

seeds: array of shape(p), optional,
initial indices of the seeds within the field if seeds==None the labels are used as initialization

labels: array of shape(self.V) initial labels, optional,
it is expected that labels take their values in a certain range (0..lmax) if Labels==None, this is not used if seeds==None and labels==None, an ewxception is raised

maxiter: int, optional,
maximal number of iterations

eps: float, optional,
increase of inertia at which convergence is declared

Returns

seeds: array of shape (p), the final seeds
label
[array of shape (self.V), the resulting field label]

J: float, inertia value

`get_E()`
To get the number of edges in the graph

`get_V()`
To get the number of vertices in the graph

`get_edges()`
To get the graph's edges

`get_field()`

`get_local_maxima`(*refdim=0*, *th=-inf*)
Look for the local maxima of one dimension (refdim) of self.field

Parameters

refdim (int) the field dimension over which the maxima are looked after
th = float, optional
threshold so that only values above th are considered

Returns

idx: array of shape (nmax)
indices of the vertices that are local maxima

depth: array of shape (nmax)
topological depth of the local maxima : depth[idx[i]] = q means that idx[i] is a q-order maximum

`get_vertices()`
To get the graph's vertices (as id)

`get_weights()`

`highest_neighbor`(*refdim=0*)
Computes the neighbor with highest field value along refdim

Parameters

refdim: int, optional,
the dimension of the field under consideration

Returns

hneighb: array of shape(self.V),
index of the neighbor with highest value

`is_connected()`

States whether self is connected or not

`kruskal()`

Creates the Minimum Spanning Tree of self using Kruskal's algo. efficient is self is sparse

Returns

K, WeightedGraph instance: the resulting MST

Notes

If self contains several connected components, will have the same number k of connected components

`left_incidence()`

Return left incidence matrix

Returns

left_incid: list
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()`

returns the set of neighbors of self as a list of arrays

`local_maxima`(*refdim=0*, *th=-inf*)

Returns all the local maxima of a field

Parameters

refdim (int) field dimension over which the maxima are looked after
th: float, optional
threshold so that only values above th are considered

Returns

depth: array of shape (nmax)
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

`main_cc()`

Returns the indexes of the vertices within the main cc

Returns

idx: array of shape (sizeof main cc)

`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

Parameters

c=0 in {0, 1, 2}, optional: index that designates the way
according to which D is normalized 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

Notes

Note that when sum_{edge[e, .] == a } D[e] = 0, nothing is performed

`opening`(*nbiter=1*)

Morphological opening of the field data. self.field is changed inplace

Parameters

nbiter: int, optional, the number of iterations required

`remove_edges`(*valid*)

Removes all the edges for which valid==0

Parameters

valid
[(self.E,) array]

`remove_trivial_edges`()

Removes trivial edges, i.e. edges that are (vv)-like self.weights and self.E are corrected accordingly

Returns

self.E (int): The number of edges

`right_incidence`()

Return right incidence matrix

Returns

right_incid: list
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_edges`(*edges*)

Sets the graph's edges

Preconditions:

- edges has a correct size
- edges take values in [1..V]

`set_euclidian`(*X*)

Compute the weights of the graph as the distances between the corresponding rows of X, which represents an embedding of self

Parameters

X array of shape (self.V, edim),
the coordinate matrix of the embedding

`set_field`(*field*)

`set_gaussian`(*X*, *sigma=0*)

Compute the weights of the graph as a gaussian function of the distance between the corresponding rows of X, which represents an embedding of self

Parameters

X array of shape (self.V, dim)
the coordinate matrix of the embedding

sigma=0, float: the parameter of the gaussian function

Notes

When sigma == 0, the following value is used: `sigma = sqrt(mean(||X[self.edges[:,  0], :]-X[self.edges[:,  1], :]||^2))`

`set_weights`(*weights*)

Set edge weights

Parameters

weights: array
array shape(self.V): edges weights

`show`(*X=None*, *ax=None*)

Plots the current graph in 2D

Parameters

X
[None or array of shape (self.V, 2)] a set of coordinates that can be used to embed the vertices in 2D. If X.shape[1]>2, a svd reduces X for display. By default, the graph is presented on a circle

ax: None or int, optional
ax handle

Returns

ax: axis handle

Notes

This should be used only for small graphs.

`subfield`(*valid*)

Returns a subfield of self, with only vertices such that valid > 0

Parameters

valid: array of shape (self.V),
nonzero for vertices to be retained

Returns

F: Field instance,
the desired subfield of self

Notes

The vertices are renumbered as [1..p] where p = sum(valid>0) when sum(valid) == 0 then None is returned

subgraph(*valid*)

Creates a subgraph with the vertices for which valid>0 and with the corresponding set of edges

Parameters

valid, array of shape (self.V): nonzero for vertices to be retained

Returns

G, WeightedGraph instance, the desired subgraph of self

Notes

The vertices are renumbered as [1..p] where p = sum(valid>0) when sum(valid==0) then None is returned

symmeterize()

Symmeterize self, modify edges and weights so that self.adjacency becomes the symmetric part of the current self.adjacency.

threshold_bifurcations(*refdim=0*, *th=-inf*)

Analysis 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 analysis

Parameters

th: float, optional,
threshold so that only values above th are considered

Returns

idx: array of shape (nlsets)
indices of the vertices that are local maxima

height: array of shape (nlsets)
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: array of shape (nlsets)
the label of the maximum which dominates each local maximum i.e. it describes the hierarchy of the local maxima

label: array of shape (self.V)
a labelling of thevertices according to their bassin

to_coo_matrix()

Return adjacency matrix as coo sparse

Returns

sp: scipy.sparse matrix instance
that encodes the adjacency matrix of 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.

Parameters

seeds: array of shape (self.V, dim)
samples: array of shape (nsamples, dim)

Notes

By default, the weights are a Gaussian function of the distance The implementation is not optimal

`voronoi_labelling`(*seed*)

Performs a voronoi labelling of the graph

Parameters

seed: array of shape (nseeds), type (np.int_),
vertices from which the cells are built

Returns

labels: array of shape (self.V) the labelling of the vertices

`ward`(*nbcluster*)

Ward's clustering of self

Parameters

nbcluster: int,
the number of desired clusters

Returns

label: array of shape (self.V)
the resulting field label

J (float): the resulting inertia

33.4 Functions

`nipy.algorithms.graph.field.`**`field_from_coo_matrix_and_data`**(*x*, *data*)

Instantiates a weighted graph from a (sparse) coo_matrix

Parameters

x: (V, V) scipy.sparse.coo_matrix instance,
the input matrix

data: array of shape (V, dim),
the field data

Returns

ifield: resulting Field instance

`nipy.algorithms.graph.field.`**`field_from_graph_and_data`**(*g*, *data*)

Instantiate a Fieldfrom a WeightedGraph plus some feature data Parameters ——— x: (V, V) scipy.sparse.coo_matrix instance,

the input matrix

data: array of shape (V, dim),
the field data

Returns

ifield: resulting field instance

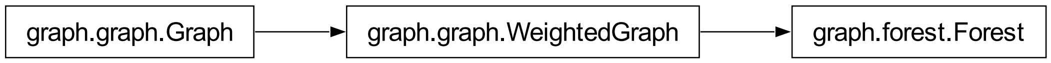
CHAPTER

THIRTYFOUR

ALGORITHMS.GRAPH.FOREST

34.1 Module: `algorithms.graph.forest`

Inheritance diagram for `nipy.algorithms.graph.forest`:



Module implements the Forest class

A Forest is a graph with a hierarchical structure. Each connected component of a forest is a tree. The main characteristic is that each node has a single parent, so that a Forest is fully characterized by a "parent" array, that defines the unique parent of each node. The directed relationships are encoded by the weight sign.

Note that some methods of WeightedGraph class (e.g. dijkstra's algorithm) require positive weights, so that they cannot work on forests in the current implementation. Specific methods (e.g. all_sidtance()) have been set instead.

Main author: Bertrand thirion, 2007-2011

34.2 `Forest`

class `nipy.algorithms.graph.forest.`**`Forest`**(*V*, *parents=None*)

Bases: *`WeightedGraph`*

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 that describe the associated 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

Attributes

V

[int] int > 0, the number of vertices

E
[int] the number of edges

parents
[(self.V,) array] the parent array

edges
[(self.E, 2) array] representing pairwise neighbors

weights
[(self.E,) array] +1/-1 for ascending/descending links

children: list
list of arrays that represents the children any node

`__init__`(*V*, *parents=None*)

Constructor

Parameters

V
[int] the number of edges of the graph

parents
[None or (V,) array] the parents of zach vertex. If `parents`==None , the parents are set to range(V), i.e. each node is its own parent, and each node is a tree

`adjacency`()

returns the adjacency matrix of the graph as a sparse coo matrix

Returns

adj: scipy.sparse matrix instance,
that encodes the adjacency matrix of self

`all_distances`(*seed=None*)

returns all the distances of the graph as a tree

Parameters

seed=None array of shape(nbseed) with valuesin [0..self.V-1]
set of vertices from which tehe distances are computed

Returns

dg: array of shape(nseed, self.V), the resulting distances

Notes

By convention infinite distances are given the distance np.inf

`anti_symmeterize`()

anti-symmeterize self, i.e. produces the graph whose adjacency matrix would be the antisymmetric part of its current adjacency matrix

`cc`()

Compte the different connected components of the graph.

Returns

label: array of shape(self.V), labelling of the vertices

`check()`

Check that self is indeed a forest, i.e. contains no loop

Returns

a boolean b=0 iff there are loops, 1 otherwise

Notes

Slow implementation, might be rewritten in C or cython

`cliques()`

Extraction of the graphe cliques these are defined using replicator dynamics equations

Returns

cliques: array of shape (self.V), type (np.int_)
labelling of the vertices according to the clique they belong to

`compact_neighb()`

returns a compact representation of self

Returns

idx: array of of shape(self.V + 1):
the positions where to find the neighbors of each node within neighb and weights

neighb: array of shape(self.E), concatenated list of neighbors
weights: array of shape(self.E), concatenated list of weights

`compute_children()`

Define the children of each node (stored in self.children)

`copy()`

returns a copy of self

`cut_redundancies()`

Returns a graph with redundant edges removed: ecah edge (ab) is present only once in the edge matrix: the correspondng weights are added.

Returns

the resulting WeightedGraph

`define_graph_attributes()`

define the edge and weights array

`degrees()`

Returns the degree of the graph vertices.

Returns

rdegree: (array, type=int, shape=(self.V,)), the right degrees
ldegree: (array, type=int, shape=(self.V,)), the left degrees

`depth_from_leaves()`

compute an index for each node: 0 for the leaves, 1 for their parents etc. and maximal for the roots.

Returns

depth: array of shape (self.V): the depth values of the vertices

`dijkstra`(*seed=0*)

Returns all the [graph] geodesic distances starting from seed x

seed (int, >-1, <self.V) or array of shape(p)
edge(s) from which the distances are computed

Returns

dg: array of shape (self.V),
the graph distance dg from ant vertex to the nearest seed

Notes

It is mandatory that the graph weights are non-negative

`floyd`(*seed=None*)

Compute all the geodesic distances starting from seeds

Parameters

seed= None: array of shape (nbseed), type np.int_
vertex indexes from which the distances are computed if seed==None, then every edge is a seed point

Returns

dg array of shape (nbseed, self.V)
the graph distance dg from each seed to any vertex

Notes

It is mandatory that the graph weights are non-negative. The algorithm proceeds by repeating Dijkstra's algo for each seed. Floyd's algo is not used (O(self.V)^3 complexity…)

`from_3d_grid`(*xyz*, *k=18*)

Sets the graph to be the topological neighbours graph of the three-dimensional coordinates set xyz, in the k-connectivity scheme

Parameters

xyz: array of shape (self.V, 3) and type np.int_,
k = 18: the number of neighbours considered. (6, 18 or 26)

Returns

E(int): the number of edges of self

`get_E`()

To get the number of edges in the graph

`get_V`()

To get the number of vertices in the graph

`get_children`(*v=-1*)

Get the children of a node/each node

Parameters

v: int, optional
a node index

Returns

children: list of int the list of children of node v (if v is provided)
a list of lists of int, the children of all nodes otherwise

`get_descendants`(*v*, *exclude_self=False*)

returns the nodes that are children of v as a list

Parameters

v: int, a node index

Returns

desc: list of int, the list of all descendant of the input node

`get_edges`()

To get the graph's edges

`get_vertices`()

To get the graph's vertices (as id)

`get_weights`()

`is_connected`()

States whether self is connected or not

`isleaf`()

Identification of the leaves of the forest

Returns

leaves: bool array of shape(self.V), indicator of the forest's leaves

`isroot`()

Returns an indicator of nodes being roots

Returns

roots, array of shape(self.V, bool), indicator of the forest's roots

`kruskal`()

Creates the Minimum Spanning Tree of self using Kruskal's algo. efficient is self is sparse

Returns

K, WeightedGraph instance: the resulting MST

Notes

If self contains several connected components, will have the same number k of connected components

`leaves_of_a_subtree`(*ids*, *custom=False*)

tests whether the given nodes are the leaves of a certain subtree

Parameters

ids: array of shape (n) that takes values in [0..self.V-1]
custom == False, boolean
if custom==true the behavior of the function is more specific - the different connected components are considered as being in a same greater tree - when a node has more than two subbranches, any subset of these children is considered as a subtree

`left_incidence()`

Return left incidence matrix

Returns

left_incid: list
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()`

returns the set of neighbors of self as a list of arrays

`main_cc()`

Returns the indexes of the vertices within the main cc

Returns

idx: array of shape (sizeof main cc)

`merge_simple_branches()`

Return a subforest, where chained branches are collapsed

Returns

sf, Forest instance, same as self, without any chain

`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

Parameters

c=0 in {0, 1, 2}, optional: index that designates the way
according to which D is normalized 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

Notes

Note that when sum_{edge[e, .] == a } D[e] = 0, nothing is performed

`propagate_upward`(*label*)

Propagation of a certain labelling from leaves to roots Assuming that label is a certain positive integer field this propagates these labels to the parents whenever the children nodes have coherent properties otherwise the parent value is unchanged

Parameters

label: array of shape(self.V)

Returns

label: array of shape(self.V)

`propagate_upward_and`(*prop*)

propagates from leaves to roots some binary property of the nodes so that prop[parents] = logical_and(prop[children])

Parameters

prop, array of shape(self.V), the input property

Returns

prop, array of shape(self.V), the output property field

`remove_edges`(*valid*)

Removes all the edges for which valid==0

Parameters

valid
[(self.E,) array]

`remove_trivial_edges`()

Removes trivial edges, i.e. edges that are (vv)-like self.weights and self.E are corrected accordingly

Returns

self.E (int): The number of edges

`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.

Returns

order: array of shape(self.V)
the order of the old vertices in the reordered graph

`right_incidence`()

Return right incidence matrix

Returns

right_incid: list
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_edges`(*edges*)

Sets the graph's edges

Preconditions:

- edges has a correct size
- edges take values in [1..V]

`set_euclidian`(*X*)

Compute the weights of the graph as the distances between the corresponding rows of X, which represents an embedding of self

Parameters

X array of shape (self.V, edim),
the coordinate matrix of the embedding

`set_gaussian`(*X*, *sigma=0*)

Compute the weights of the graph as a gaussian function of the distance between the corresponding rows of X, which represents an embedding of self

Parameters

X array of shape (self.V, dim)
the coordinate matrix of the embedding

sigma=0, float: the parameter of the gaussian function

Notes

When sigma == 0, the following value is used: `sigma = sqrt(mean(||X[self.edges[:, 0], :]-X[self.edges[:, 1], :]||^2))`

`set_weights`(*weights*)

Set edge weights

Parameters

weights: array
array shape(self.V): edges weights

`show`(*X=None*, *ax=None*)

Plots the current graph in 2D

Parameters

X
[None or array of shape (self.V, 2)] a set of coordinates that can be used to embed the vertices in 2D. If X.shape[1]>2, a svd reduces X for display. By default, the graph is presented on a circle

ax: None or int, optional
ax handle

Returns

ax: axis handle

Notes

This should be used only for small graphs.

`subforest`(*valid*)

Creates a subforest with the vertices for which valid > 0

Parameters

valid: array of shape (self.V): indicator of the selected nodes

Returns

subforest: a new forest instance, with a reduced set of nodes

Notes

The children of deleted vertices become their own parent

`subgraph`(*valid*)

Creates a subgraph with the vertices for which valid>0 and with the corresponding set of edges

Parameters

valid, array of shape (self.V): nonzero for vertices to be retained

Returns

G, WeightedGraph instance, the desired subgraph of self

Notes

The vertices are renumbered as [1..p] where p = sum(valid>0) when sum(valid==0) then None is returned

`symmeterize()`

Symmeterize self, modify edges and weights so that self.adjacency becomes the symmetric part of the current self.adjacency.

`to_coo_matrix()`

Return adjacency matrix as coo sparse

Returns

sp: scipy.sparse matrix instance
that encodes the adjacency matrix of self

`tree_depth()`

Returns the number of hierarchical levels in the tree

`voronoi_diagram`(*seeds*, *samples*)

Defines the graph as the Voronoi diagram (VD) that links the seeds. The VD is defined using the sample points.

Parameters

seeds: array of shape (self.V, dim)
samples: array of shape (nsamples, dim)

Notes

By default, the weights are a Gaussian function of the distance The implementation is not optimal

`voronoi_labelling`(*seed*)

Performs a voronoi labelling of the graph

Parameters

seed: array of shape (nseeds), type (np.int_),
vertices from which the cells are built

Returns

labels: array of shape (self.V) the labelling of the vertices

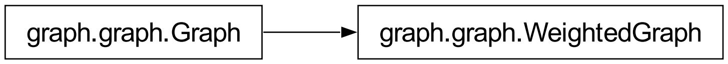
CHAPTER

THIRTYFIVE

ALGORITHMS.GRAPH.GRAPH

35.1 Module: `algorithms.graph.graph`

Inheritance diagram for `nipy.algorithms.graph.graph`:



This module implements two graph classes:

Graph: basic topological graph, i.e. vertices and edges. This kind of object only has topological properties

WeightedGraph (Graph): also has a value associated with edges, called weights, that are used in some computational procedures (e.g. path length computation). Importantly these objects are equivalent to square sparse matrices, which is used to perform certain computations.

This module also provides several functions to instantiate WeightedGraphs from data: - k nearest neighbours (where samples are rows of a 2D-array) - epsilon-neighbors (where sample rows of a 2D-array) - representation of the neighbors on a 3d grid (6-, 18- and 26-neighbors) - Minimum Spanning Tree (where samples are rows of a 2D-array)

Author: Bertrand Thirion, 2006–2011

35.2 Classes

35.2.1 `Graph`

class `nipy.algorithms.graph.graph.`**`Graph`**(*V*, *E=0*, *edges=None*)

Bases: `object`

Basic topological (non-weighted) directed Graph class

Member variables:

- V (int > 0): the number of vertices
- E (int >= 0): the number of edges

Properties:

- vertices (list, type=int, shape=(V,)) vertices id
- edges (list, type=int, shape=(E,2)): edges as vertices id tuples

`__init__`(*V*, *E=0*, *edges=None*)

Constructor

Parameters

V
[int] the number of vertices

E
[int, optional] the number of edges

edges
[None or shape (E, 2) array, optional] edges of graph

`adjacency`()

returns the adjacency matrix of the graph as a sparse coo matrix

Returns

adj: scipy.sparse matrix instance,
that encodes the adjacency matrix of self

`cc`()

Compte the different connected components of the graph.

Returns

label: array of shape(self.V), labelling of the vertices

`degrees`()

Returns the degree of the graph vertices.

Returns

rdegree: (array, type=int, shape=(self.V,)), the right degrees
ldegree: (array, type=int, shape=(self.V,)), the left degrees

`get_E`()

To get the number of edges in the graph

`get_V`()

To get the number of vertices in the graph

`get_edges`()

To get the graph's edges

`get_vertices`()

To get the graph's vertices (as id)

`main_cc`()

Returns the indexes of the vertices within the main cc

Returns

idx: array of shape (sizeof main cc)

`set_edges`(*edges*)

Sets the graph's edges

Preconditions:

- edges has a correct size
- edges take values in [1..V]

`show`(*ax=None*)

Shows the graph as a planar one.

Parameters

ax, axis handle

Returns

ax, axis handle

`to_coo_matrix`()

Return adjacency matrix as coo sparse

Returns

sp: scipy.sparse matrix instance,
that encodes the adjacency matrix of self

35.2.2 WeightedGraph

class `nipy.algorithms.graph.graph.WeightedGraph`(*V*, *edges=None*, *weights=None*)

Bases: *Graph*

Basic weighted, directed graph class

Member variables:

- V (int): the number of vertices
- E (int): the number of edges

Methods

- vertices (list, type=int, shape=(V,)): vertices id
- edges (list, type=int, shape=(E,2)): edges as vertices id tuples
- weights (list, type=int, shape=(E,)): weights / lengths of the graph's edges

`__init__`(*V*, *edges=None*, *weights=None*)

Constructor

Parameters

V
[int] (int > 0) the number of vertices

edges
[(E, 2) array, type int] edges of the graph

weights
[(E, 2) array, type=int] weights/lengths of the edges

`adjacency()`

returns the adjacency matrix of the graph as a sparse coo matrix

Returns

adj: scipy.sparse matrix instance,
that encodes the adjacency matrix of self

`anti_symmeterize()`

anti-symmeterize self, i.e. produces the graph whose adjacency matrix would be the antisymmetric part of its current adjacency matrix

`cc()`

Compte the different connected components of the graph.

Returns

label: array of shape(self.V), labelling of the vertices

`cliques()`

Extraction of the graphe cliques these are defined using replicator dynamics equations

Returns

cliques: array of shape (self.V), type (np.int_)
labelling of the vertices according to the clique they belong to

`compact_neighb()`

returns a compact representation of self

Returns

idx: array of of shape(self.V + 1):
the positions where to find the neighbors of each node within neighb and weights

neighb: array of shape(self.E), concatenated list of neighbors
weights: array of shape(self.E), concatenated list of weights

`copy()`

returns a copy of self

`cut_redundancies()`

Returns a graph with redundant edges removed: ecah edge (ab) is present only once in the edge matrix: the correspondng weights are added.

Returns

the resulting WeightedGraph

`degrees()`

Returns the degree of the graph vertices.

Returns

rdegree: (array, type=int, shape=(self.V,)), the right degrees
ldegree: (array, type=int, shape=(self.V,)), the left degrees

`dijkstra(`*seed=0*`)`

Returns all the [graph] geodesic distances starting from seed x

seed (int, >-1, <self.V) or array of shape(p)
edge(s) from which the distances are computed

Returns

dg: array of shape (self.V),
the graph distance dg from ant vertex to the nearest seed

Notes

It is mandatory that the graph weights are non-negative

`floyd`(*seed=None*)

Compute all the geodesic distances starting from seeds

Parameters

seed= None: array of shape (nbseed), type np.int_
vertex indexes from which the distances are computed if seed==None, then every edge is a seed point

Returns

dg array of shape (nbseed, self.V)
the graph distance dg from each seed to any vertex

Notes

It is mandatory that the graph weights are non-negative. The algorithm proceeds by repeating Dijkstra's algo for each seed. Floyd's algo is not used (O(self.V)^3 complexity…)

`from_3d_grid`(*xyz*, *k=18*)

Sets the graph to be the topological neighbours graph of the three-dimensional coordinates set xyz, in the k-connectivity scheme

Parameters

xyz: array of shape (self.V, 3) and type np.int_,
k = 18: the number of neighbours considered. (6, 18 or 26)

Returns

E(int): the number of edges of self

`get_E`()

To get the number of edges in the graph

`get_V`()

To get the number of vertices in the graph

`get_edges`()

To get the graph's edges

`get_vertices`()

To get the graph's vertices (as id)

`get_weights`()

`is_connected`()

States whether self is connected or not

`kruskal()`

Creates the Minimum Spanning Tree of self using Kruskal's algo. efficient is self is sparse

Returns

K, WeightedGraph instance: the resulting MST

Notes

If self contains several connected components, will have the same number k of connected components

`left_incidence()`

Return left incidence matrix

Returns

left_incid: list
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()`

returns the set of neighbors of self as a list of arrays

`main_cc()`

Returns the indexes of the vertices within the main cc

Returns

idx: array of shape (sizeof main cc)

`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

Parameters

c=0 in {0, 1, 2}, optional: index that designates the way
according to which D is normalized 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

Notes

Note that when sum_{edge[e, .] == a } D[e] = 0, nothing is performed

`remove_edges(valid)`

Removes all the edges for which valid==0

Parameters

valid
[(self.E,) array]

`remove_trivial_edges()`

Removes trivial edges, i.e. edges that are (vv)-like self.weights and self.E are corrected accordingly

Returns

self.E (int): The number of edges

`right_incidence()`

Return right incidence matrix

Returns

right_incid: list
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_edges(`*edges*`)`

Sets the graph's edges

Preconditions:

- edges has a correct size
- edges take values in [1..V]

`set_euclidian(`*X*`)`

Compute the weights of the graph as the distances between the corresponding rows of X, which represents an embedding of self

Parameters

X array of shape (self.V, edim),
the coordinate matrix of the embedding

`set_gaussian(`*X*, *sigma=0*`)`

Compute the weights of the graph as a gaussian function of the distance between the corresponding rows of X, which represents an embedding of self

Parameters

X array of shape (self.V, dim)
the coordinate matrix of the embedding

sigma=0, float: the parameter of the gaussian function

Notes

When sigma == 0, the following value is used: `sigma = sqrt(mean(||X[self.edges[:, 0], :]-X[self.edges[:, 1], :]||^2))`

`set_weights(`*weights*`)`

Set edge weights

Parameters

weights: array
array shape(self.V): edges weights

`show(`*X=None*, *ax=None*`)`

Plots the current graph in 2D

Parameters

X
[None or array of shape (self.V, 2)] a set of coordinates that can be used to embed the vertices in 2D. If X.shape[1]>2, a svd reduces X for display. By default, the graph is presented on a circle

ax: None or int, optional
ax handle

Returns

ax: axis handle

Notes

This should be used only for small graphs.

`subgraph`(*valid*)

Creates a subgraph with the vertices for which valid>0 and with the corresponding set of edges

Parameters

valid, array of shape (self.V): nonzero for vertices to be retained

Returns

G, WeightedGraph instance, the desired subgraph of self

Notes

The vertices are renumbered as [1..p] where p = sum(valid>0) when sum(valid==0) then None is returned

`symmeterize`()

Symmeterize self, modify edges and weights so that self.adjacency becomes the symmetric part of the current self.adjacency.

`to_coo_matrix`()

Return adjacency matrix as coo sparse

Returns

sp: scipy.sparse matrix instance
that encodes the adjacency matrix of 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.

Parameters

seeds: array of shape (self.V, dim)
samples: array of shape (nsamples, dim)

Notes

By default, the weights are a Gaussian function of the distance The implementation is not optimal

`voronoi_labelling`(*seed*)

Performs a voronoi labelling of the graph

Parameters

seed: array of shape (nseeds), type (np.int_),
vertices from which the cells are built

Returns

labels: array of shape (self.V) the labelling of the vertices

35.3 Functions

`nipy.algorithms.graph.graph.`**`complete_graph`**`(`*n*`)`

returns a complete graph with n vertices

`nipy.algorithms.graph.graph.`**`concatenate_graphs`**`(`*G1*, *G2*`)`

Returns the concatenation of the graphs G1 and G2 It is thus assumed that the vertices of G1 and G2 represent disjoint sets

Parameters

G1, G2: the two WeightedGraph instances to be concatenated

Returns

G, WeightedGraph, the concatenated graph

Notes

This implies that the vertices of G corresponding to G2 are labeled [G1.V .. G1.V+G2.V]

`nipy.algorithms.graph.graph.`**`eps_nn`**`(`*X*, *eps=1.0*`)`

Returns the eps-nearest-neighbours graph of the data

Parameters

X, array of shape (n_samples, n_features), input data
eps, float, optional: the neighborhood width

Returns

the resulting graph instance

`nipy.algorithms.graph.graph.`**`graph_3d_grid`**`(`*xyz*, *k=18*`)`

Utility that computes the six neighbors on a 3d grid

Parameters

xyz: array of shape (n_samples, 3); grid coordinates of the points
k: neighboring system, equal to 6, 18, or 26

Returns

i, j, d 3 arrays of shape (E),
where E is the number of edges in the resulting graph (i, j) represent the edges, d their weights

`nipy.algorithms.graph.graph.`**`knn`**`(`*X*, *k=1*`)`

returns the k-nearest-neighbours graph of the data

Parameters

X, array of shape (n_samples, n_features): the input data
k, int, optional: is the number of neighbours considered

Returns

the corresponding WeightedGraph instance

Notes

The knn system is symmeterized: if (ab) is one of the edges then (ba) is also included

`nipy.algorithms.graph.graph.`**`lil_cc`**`(`*lil*`)`

Returns the connected components of a graph represented as a list of lists

Parameters

lil: a list of list representing the graph neighbors

Returns

label a vector of shape len(lil): connected components labelling

Notes

Dramatically slow for non-sparse graphs

`nipy.algorithms.graph.graph.`**`mst`**`(`*X*`)`

Returns the WeightedGraph that is the minimum Spanning Tree of X

Parameters

X: data array, of shape(n_samples, n_features)

Returns

the corresponding WeightedGraph instance

`nipy.algorithms.graph.graph.`**`wgraph_from_3d_grid`**`(`*xyz*, *k=18*`)`

Create graph as the set of topological neighbours of the three-dimensional coordinates set xyz, in the k-connectivity scheme

Parameters

xyz: array of shape (nsamples, 3) and type np.int_,
k = 18: the number of neighbours considered. (6, 18 or 26)

Returns

the WeightedGraph instance

`nipy.algorithms.graph.graph.`**`wgraph_from_adjacency`**`(`*x*`)`

Instantiates a weighted graph from a square 2D array

Parameters

x: 2D array instance, the input array

Returns

wg: WeightedGraph instance

`nipy.algorithms.graph.graph.`**`wgraph_from_coo_matrix`**`(`*x*`)`

Instantiates a weighted graph from a (sparse) coo_matrix

Parameters

x: scipy.sparse.coo_matrix instance, the input matrix

Returns

wg: WeightedGraph instance

CHAPTER

THIRTYSIX

ALGORITHMS.GROUP.PARCEL_ANALYSIS

36.1 Module: `algorithms.group.parcel_analysis`

Inheritance diagram for `nipy.algorithms.group.parcel_analysis`:

Parcel-based group analysis of multi-subject image data.

Routines implementing Bayesian inference on group-level effects assumed to be constant within given brain parcels. The model accounts for both estimation errors and localization uncertainty in reference space of first-level images.

See:

Keller, Merlin et al (2008). Dealing with Spatial Normalization Errors in fMRI Group Inference using Hierarchical Modeling. *Statistica Sinica*; 18(4).

Keller, Merlin et al (2009). Anatomically Informed Bayesian Model Selection for fMRI Group Data Analysis. *In MICCAI'09, Lecture Notes in Computer Science*; 5762:450–457.

Roche, Alexis (2012). OHBM'12 talk, slides at: https://sites.google.com/site/alexisroche/slides/Talk_Beijing12.pdf

36.2 `ParcelAnalysis`

class `nipy.algorithms.group.parcel_analysis.`**`ParcelAnalysis`**(*con_imgs*, *parcel_img*, *parcel_info=None*, *msk_img=None*, *vcon_imgs=None*, *design_matrix=None*, *cvect=None*, *fwhm=8*, *smooth_method='default'*, *res_path=None*, *write_smoothed_images=False*)

Bases: `object`

`__init__`(*con_imgs*, *parcel_img*, *parcel_info=None*, *msk_img=None*, *vcon_imgs=None*, *design_matrix=None*, *cvect=None*, *fwhm=8*, *smooth_method='default'*, *res_path=None*, *write_smoothed_images=False*)

Bayesian parcel-based analysis.

Given a sequence of independent images registered to a common space (for instance, a set of contrast images from a first-level fMRI analysis), perform a second-level analysis assuming constant effects throughout parcels defined from a given label image in reference space. Specifically, a model of the following form is assumed:

Y = X * beta + variability,

where Y denotes the input image sequence, X is a design matrix, and beta are parcel-wise parameter vectors. The algorithm computes the Bayesian posterior probability of beta in each parcel using an expectation propagation scheme.

Parameters

con_imgs: sequence of nipy-like images
Images input to the group analysis.

parcel_img: nipy-like image
Label image where each label codes for a parcel.

parcel_info: sequence of arrays, optional
A sequence of two arrays with same length equal to the number of distinct parcels consistently with the *parcel_img* argument. The first array gives parcel names and the second, parcel values, i.e., corresponding intensities in the associated parcel image. By default, parcel values are taken as *np.unique(parcel_img.get_fdata())* and parcel names are these values converted to strings.

msk_img: nipy-like image, optional
Binary mask to restrict analysis. By default, analysis is carried out on all parcels with nonzero value.

vcon_imgs: sequence of nipy-like images, optional
First-level variance estimates corresponding to *con_imgs*. This is useful if the input images are "noisy". By default, first-level variances are assumed to be zero.

design_matrix: array, optional
If None, a one-sample analysis model is used. Otherwise, an array with shape (n, p) where *n* matches the number of input scans, and *p* is the number of regressors.

cvect: array, optional
Contrast vector of interest. The method makes an inference on the contrast defined as the dot product cvect'*beta, where beta are the unknown parcel-wise effects. If None, *cvect* is assumed to be np.array((1,)). However, the *cvect* argument is mandatory if *design_matrix* is provided.

fwhm: float, optional
A parameter that represents the localization uncertainty in reference space in terms of the full width at half maximum of an isotropic Gaussian kernel.

smooth_method: str, optional
One of 'default' and 'spm'. Setting *smooth_method=spm* results in simply smoothing the input images using a Gaussian kernel, while the default method involves more complex smoothing in order to propagate spatial uncertainty into the inference process.

res_path: str, optional
An existing path to write output images. If None, no output is written.

write_smoothed_images: bool, optional
Specify whether smoothed images computed throughout the inference process are to be

written on disk in *res_path*.

`dump_results`(*path=None*)

Save parcel analysis information in NPZ file.

`parcel_maps`(*full_res=True*)

Compute parcel-based posterior contrast means and positive contrast probabilities.

Parameters

full_res: boolean
If True, the output images will be at the same resolution as the parcel image. Otherwise, resolution will match the first-level images.

Returns

pmap_mu_img: nipy image
Image of posterior contrast means for each parcel.

pmap_prob_img: nipy image
Corresponding image of posterior probabilities of positive contrast.

`t_map`()

Compute voxel-wise t-statistic map. This map is different from what you would get from an SPM-style mass univariate analysis because the method accounts for both spatial uncertainty in reference space and possibly errors on first-level inputs (if variance images are provided).

Returns

tmap_img: nipy image
t-statistic map.

`nipy.algorithms.group.parcel_analysis.parcel_analysis`(*con_imgs*, *parcel_img*, *msk_img=None*, *vcon_imgs=None*, *design_matrix=None*, *cvect=None*, *fwhm=8*, *smooth_method='default'*, *res_path=None*)

Helper function for Bayesian parcel-based analysis.

Given a sequence of independent images registered to a common space (for instance, a set of contrast images from a first-level fMRI analysis), perform a second-level analysis assuming constant effects throughout parcels defined from a given label image in reference space. Specifically, a model of the following form is assumed:

Y = X * beta + variability,

where Y denotes the input image sequence, X is a design matrix, and beta are parcel-wise parameter vectors. The algorithm computes the Bayesian posterior probability of cvect'*beta, where cvect is a given contrast vector, in each parcel using an expectation propagation scheme.

Parameters

con_imgs: sequence of nipy-like images
Images input to the group analysis.

parcel_img: nipy-like image
Label image where each label codes for a parcel.

msk_img: nipy-like image, optional
Binary mask to restrict analysis. By default, analysis is carried out on all parcels with nonzero value.

vcon_imgs: sequence of nipy-like images, optional
First-level variance estimates corresponding to *con_imgs*. This is useful if the input images are “noisy”. By default, first-level variances are assumed to be zero.

design_matrix: array, optional
If None, a one-sample analysis model is used. Otherwise, an array with shape (n, p) where *n* matches the number of input scans, and *p* is the number of regressors.

cvect: array, optional
Contrast vector of interest. The method makes an inference on the contrast defined as the dot product cvect'*beta, where beta are the unknown parcel-wise effects. If None, *cvect* is assumed to be np.array((1,)). However, the *cvect* argument is mandatory if *design_matrix* is provided.

fwhm: float, optional
A parameter that represents the localization uncertainty in reference space in terms of the full width at half maximum of an isotropic Gaussian kernel.

smooth_method: str, optional
One of 'default' and 'spm'. Setting *smooth_method=spm* results in simply smoothing the input images using a Gaussian kernel, while the default method involves more complex smoothing in order to propagate spatial uncertainty into the inference process.

res_path: str, optional
An existing path to write output images. If None, no output is written.

Returns

pmap_mu_img: nipy image
Image of posterior contrast means for each parcel.

pmap_prob_img: nipy image
Corresponding image of posterior probabilities of positive contrast.

CHAPTER

THIRTYSEVEN

ALGORITHMS.INTERPOLATION

37.1 Module: `algorithms.interpolation`

Inheritance diagram for `nipy.algorithms.interpolation`:

Image interpolators using ndimage.

37.2 `ImageInterpolator`

class `nipy.algorithms.interpolation.`**`ImageInterpolator`**(*image*, *order=3*, *mode='constant'*, *cval=0.0*)

Bases: `object`

Interpolate Image instance at arbitrary points in world space

The resampling is done with `scipy.ndimage`.

`__init__`(*image*, *order=3*, *mode='constant'*, *cval=0.0*)

Parameters

image
[Image] Image to be interpolated.

order
[int, optional] order of spline interpolation as used in `scipy.ndimage`. Default is 3.

mode
[str, optional] Points outside the boundaries of the input are filled according to the given mode (‘constant’, ‘nearest’, ‘reflect’ or ‘wrap’). Default is ‘constant’.

cval
[scalar, optional] Value used for points outside the boundaries of the input if mode=’constant’. Default is 0.0.

`evaluate`(*points*)

Resample image at points in world space

Parameters

points
[array] values in self.image.coordmap.output_coords. Each row is a point.

Returns

V
[ndarray] interpolator of self.image evaluated at points

`property mode`

Mode is read-only

`n_prepad_if_needed = 12`

`property order`

Order is read-only

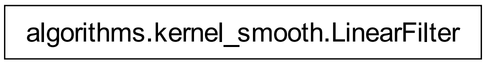
CHAPTER

THIRTYEIGHT

ALGORITHMS.KERNEL_SMOOTH

38.1 Module: `algorithms.kernel_smooth`

Inheritance diagram for `nipy.algorithms.kernel_smooth`:



Linear filter(s). For the moment, only a Gaussian smoothing filter

38.2 Class

38.3 `LinearFilter`

class `nipy.algorithms.kernel_smooth.`**`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
[`CoordinateMap`]

shape
[sequence]

fwhm
[float, optional] fwhm for Gaussian kernel, default is 6.0

scale
[float, optional] scaling to apply to data after smooth, default 1.0

location
[float] offset to apply to data after smooth and scaling, default 0

cov
[None or array, optional] Covariance matrix

`normalization = 'l1sum'`

`smooth`(*inimage*, *clean=False*, *is_fft=False*)

Apply smoothing to *inimage*

Parameters

inimage
[`Image`] The image to be smoothed. Should be 3D.

clean
[bool, optional] Should we call `nan_to_num` on the data before smoothing?

is_fft
[bool, optional] Has the data already been fft'd?

Returns

s_image
[*Image*] New image, with smoothing applied

38.4 Functions

`nipy.algorithms.kernel_smooth.fwhm2sigma`(*fwhm*)

Convert a FWHM value to sigma in a Gaussian kernel.

Parameters

fwhm
[array-like] FWHM value or values

Returns

sigma
[array or float] sigma values corresponding to *fwhm* values

Examples

```
>>> sigma = fwhm2sigma(6)
>>> sigmae = fwhm2sigma([6, 7, 8])
>>> sigma == sigmae[0]
True
```

`nipy.algorithms.kernel_smooth.sigma2fwhm`(*sigma*)

Convert a sigma in a Gaussian kernel to a FWHM value

Parameters

sigma
[array-like] sigma value or values

Returns

fwhm
[array or float] fwhm values corresponding to *sigma* values

Examples

```
>>> fwhm = sigma2fwhm(3)
>>> fwhms = sigma2fwhm([3, 4, 5])
>>> fwhm == fwhms[0]
True
```

CHAPTER THIRTYNINE

ALGORITHMS.OPTIMIZE

39.1 Module: `algorithms.optimize`

`nipy.algorithms.optimize.`**`fmin_steepest`**(*f*, *x0*, *fprime=None*, *xtol=0.0001*, *ftol=0.0001*, *maxiter=None*, *epsilon=1.4901161193847656e-08*, *callback=None*, *disp=True*)

Minimize a function using a steepest gradient descent algorithm. This complements the collection of minimization routines provided in scipy.optimize. Steepest gradient iterations are cheaper than in the conjugate gradient or Newton methods, hence convergence may sometimes turn out faster algthough more iterations are typically needed.

Parameters

f
[callable] Function to be minimized

x0
[array] Starting point

fprime
[callable] Function that computes the gradient of f

xtol
[float] Relative tolerance on step sizes in line searches

ftol
[float] Relative tolerance on function variations

maxiter
[int] Maximum number of iterations

epsilon
[float or ndarray] If fprime is approximated, use this value for the step

size (can be scalar or vector).
callback
[callable] Optional function called after each iteration is complete

disp
[bool] Print convergence message if True

Returns

x
[array] Gradient descent fix point, local minimizer of f

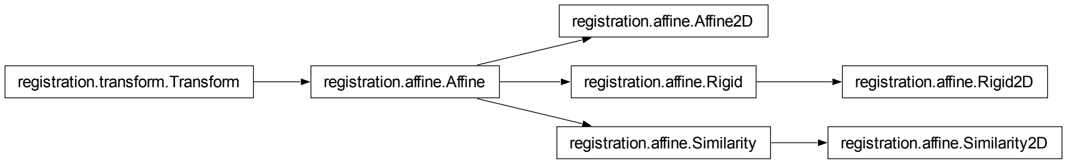
CHAPTER

FORTY

ALGORITHMS.REGISTRATION.AFFINE

40.1 Module: `algorithms.registration.affine`

Inheritance diagram for `nipy.algorithms.registration.affine`:


registration.transform.Transform
registration.affine.Affine
registration.affine.Affine2D
registration.affine.Rigid
registration.affine.Rigid2D
registration.affine.Similarity
registration.affine.Similarity2D

40.2 Classes

40.2.1 `Affine`

class `nipy.algorithms.registration.affine.`**`Affine`**(*array=None*, *radius=100*)

Bases: `Transform`

`__init__`(*array=None*, *radius=100*)

`apply`(*xyz*)

`as_affine`(*dtype='double'*)

`compose`(*other*)

Compose this transform onto another

Parameters

other
[Transform] transform that we compose onto

Returns

composed_transform
[Transform] a transform implementing the composition of self on *other*

`copy()`

`from_matrix44(`*aff*`)`

Convert a 4x4 matrix describing an affine transform into a 12-sized vector of natural affine parameters: translation, rotation, log-scale, pre-rotation (to allow for shearing when combined with non-unitary scales). In case the transform has a negative determinant, set the *_direct* attribute to False.

`inv()`

Return the inverse affine transform.

`property is_direct`

`property param`

`param_inds = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`

`property pre_rotation`

`property precond`

`property rotation`

`property scaling`

`property translation`

40.2.2 Affine2D

`class nipy.algorithms.registration.affine.Affine2D(`*array=None*, *radius=100*`)`

Bases: *Affine*

`__init__(`*array=None*, *radius=100*`)`

`apply(`*xyz*`)`

`as_affine(`*dtype='double'*`)`

`compose(`*other*`)`

Compose this transform onto another

Parameters

other
[Transform] transform that we compose onto

Returns

composed_transform
[Transform] a transform implementing the composition of self on *other*

`copy()`

`from_matrix44(`*aff*`)`

Convert a 4x4 matrix describing an affine transform into a 12-sized vector of natural affine parameters: translation, rotation, log-scale, pre-rotation (to allow for shearing when combined with non-unitary scales). In case the transform has a negative determinant, set the *_direct* attribute to False.

inv()

Return the inverse affine transform.

property is_direct

property param

param_inds = [0, 1, 5, 6, 7, 11]

property pre_rotation

property precond

property rotation

property scaling

property translation

40.2.3 Rigid

class nipy.algorithms.registration.affine.**Rigid**(*array=None*, *radius=100*)

Bases: *Affine*

__init__(*array=None*, *radius=100*)

apply(*xyz*)

as_affine(*dtype='double'*)

compose(*other*)

Compose this transform onto another

Parameters

other
[Transform] transform that we compose onto

Returns

composed_transform
[Transform] a transform implementing the composition of self on *other*

copy()

from_matrix44(*aff*)

Convert a 4x4 matrix describing a rigid transform into a 12-sized vector of natural affine parameters: translation, rotation, log-scale, pre-rotation (to allow for pre-rotation when combined with non-unitary scales). In case the transform has a negative determinant, set the *_direct* attribute to False.

inv()

Return the inverse affine transform.

property is_direct

property param

param_inds = [0, 1, 2, 3, 4, 5]

property pre_rotation

property precond

property rotation

property scaling

property translation

40.2.4 Rigid2D

class nipy.algorithms.registration.affine.**Rigid2D**(*array=None*, *radius=100*)

Bases: *Rigid*

__init__(*array=None*, *radius=100*)

apply(*xyz*)

as_affine(*dtype='double'*)

compose(*other*)

Compose this transform onto another

Parameters

other
[Transform] transform that we compose onto

Returns

composed_transform
[Transform] a transform implementing the composition of self on *other*

copy()

from_matrix44(*aff*)

Convert a 4x4 matrix describing a rigid transform into a 12-sized vector of natural affine parameters: translation, rotation, log-scale, pre-rotation (to allow for pre-rotation when combined with non-unitary scales). In case the transform has a negative determinant, set the *_direct* attribute to False.

inv()

Return the inverse affine transform.

property is_direct

property param

param_inds = [0, 1, 5]

property pre_rotation

property precond

property rotation

property scaling

property translation

40.2.5 Similarity

class nipy.algorithms.registration.affine.**Similarity**(*array=None*, *radius=100*)

Bases: *Affine*

__init__(*array=None*, *radius=100*)

apply(*xyz*)

as_affine(*dtype='double'*)

compose(*other*)

Compose this transform onto another

Parameters

other
[Transform] transform that we compose onto

Returns

composed_transform
[Transform] a transform implementing the composition of self on *other*

copy()

from_matrix44(*aff*)

Convert a 4x4 matrix describing a similarity transform into a 12-sized vector of natural affine parameters: translation, rotation, log-scale, pre-rotation (to allow for pre-rotation when combined with non-unitary scales). In case the transform has a negative determinant, set the *_direct* attribute to False.

inv()

Return the inverse affine transform.

property is_direct

property param

param_inds = [0, 1, 2, 3, 4, 5, 6]

property pre_rotation

property precond

property rotation

property scaling

property translation

40.2.6 Similarity2D

class nipy.algorithms.registration.affine.**Similarity2D**(*array=None*, *radius=100*)

Bases: *Similarity*

__init__(*array=None*, *radius=100*)

apply(*xyz*)

as_affine(*dtype='double'*)

compose(*other*)

Compose this transform onto another

Parameters

other
[Transform] transform that we compose onto

Returns

composed_transform
[Transform] a transform implementing the composition of self on *other*

copy()

from_matrix44(*aff*)

Convert a 4x4 matrix describing a similarity transform into a 12-sized vector of natural affine parameters: translation, rotation, log-scale, pre-rotation (to allow for pre-rotation when combined with non-unitary scales). In case the transform has a negative determinant, set the *_direct* attribute to False.

inv()

Return the inverse affine transform.

property is_direct

property param

param_inds = [0, 1, 5, 6]

property pre_rotation

property precond

property rotation

property scaling

property translation

40.3 Functions

`nipy.algorithms.registration.affine.inverse_affine(affine)`

`nipy.algorithms.registration.affine.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 the vector of log-scales, then all components of (p/pc) are roughly comparable to the translation component.

To that end, we use a *radius* parameter which represents the 'typical size' of the object being registered. This is used to reformat the parameter vector (translation+rotation+scaling+pre-rotation) so that each element roughly represents a variation in mm.

`nipy.algorithms.registration.affine.rotation_mat2vec(R)`

Rotation vector from rotation matrix *R*

Parameters

R
[(3,3) array-like] Rotation matrix

Returns

vec
[(3,) array] Rotation vector, where norm of *vec* is the angle `theta`, and the axis of rotation is given by `vec / theta`

`nipy.algorithms.registration.affine.rotation_vec2mat(r)`

The rotation matrix is given by the Rodrigues formula:

R = Id + sin(theta)*Sn + (1-cos(theta))*Sn^2

with:

0 -nz ny

Sn = nz 0 -nx

-ny nx 0

where n = r / ||r||

In case the angle ||r|| is very small, the above formula may lead to numerical instabilities. We instead use a Taylor expansion around theta=0:

R = I + sin(theta)/tetha Sr + (1-cos(theta))/teta2 Sr^2

leading to:

R = I + (1-theta2/6)*Sr + (1/2-theta2/24)*Sr^2

To avoid numerical instabilities, an upper threshold is applied to the angle. It is chosen to be a multiple of 2*pi, hence the resulting rotation is then the identity matrix. This strategy warrants that the output matrix is a continuous function of the input vector.

`nipy.algorithms.registration.affine.slices2aff(slices)`

Return affine from start, step of sequence *slices* of slice objects

Parameters

slices
[sequence of slice objects]

Returns

aff
[ndarray] If `N = len(slices)` then affine is shape (N+1, N+1) with diagonal given by the `step` attribute of the slice objects (where None corresponds to 1), and the *:N* elements in the last column are given by the `start` attribute of the slice objects

Examples

```
>>> slices2aff([slice(None), slice(None)])
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> slices2aff([slice(2, 3, 4), slice(3, 4, 5), slice(4, 5, 6)])
array([[ 4.,  0.,  0.,  2.],
       [ 0.,  5.,  0.,  3.],
       [ 0.,  0.,  6.,  4.],
       [ 0.,  0.,  0.,  1.]])
```

`nipy.algorithms.registration.affine.`**`subgrid_affine`**`(`*affine*, *slices*`)`

Return dot prodoct of *affine* and affine resulting from *slices*

Parameters

affine
[array-like] Affine to apply on right of affine resulting from *slices*

slices
[sequence of slice objects] Slices generating (N+1, N+1) affine from `slices2aff`, where `N = len(slices)`

Returns

aff
[ndarray] result of `np.dot(affine, slice_affine)` where `slice_affine` is affine resulting from `slices2aff(slices)`.

Raises

ValueError
[if the `slice_affine` contains non-integer values]

`nipy.algorithms.registration.affine.`**`threshold`**`(`*x*, *th*`)`

`nipy.algorithms.registration.affine.`**`to_matrix44`**`(`*t*`)`

t is a vector of affine transformation parameters with size at least 6.

size < 6 ==> error size == 6 ==> t is interpreted as translation + rotation size == 7 ==> t is interpreted as translation + rotation + isotropic scaling 7 < size < 12 ==> error size >= 12 ==> t is interpreted as translation + rotation + scaling + pre-rotation

CHAPTER

FORTYONE

ALGORITHMS.REGISTRATION.CHAIN_TRANSFORM

41.1 Module: `algorithms.registration.chain_transform`

Inheritance diagram for `nipy.algorithms.registration.chain_transform`:

Chain transforms

41.2 `ChainTransform`

class `nipy.algorithms.registration.chain_transform.`**`ChainTransform`**(*optimizable*, *pre=None*, *post=None*)

Bases: `object`

`__init__`(*optimizable*, *pre=None*, *post=None*)

Create chain transform instance

Parameters

optimizable
[array or Transform] Transform that we are optimizing. If this is an array, then assume it's an affine matrix.

pre
[None or array or Transform, optional] If not None, a transform that should be applied to points before applying the *optimizable* transform. If an array, then assume it's an affine matrix.

post
[None or Transform, optional] If not None, a transform that should be applied to points after applying any *pre* transform, and then the *optimizable* transform. If an array, assume it's an affine matrix

`apply`(*pts*)

Apply full transformation to points *pts*

If there are N points, then *pts* will be N by 3

Parameters

pts
[array-like] array of points

Returns

transformed_pts
[array] N by 3 array of transformed points

`property param`

get/set param

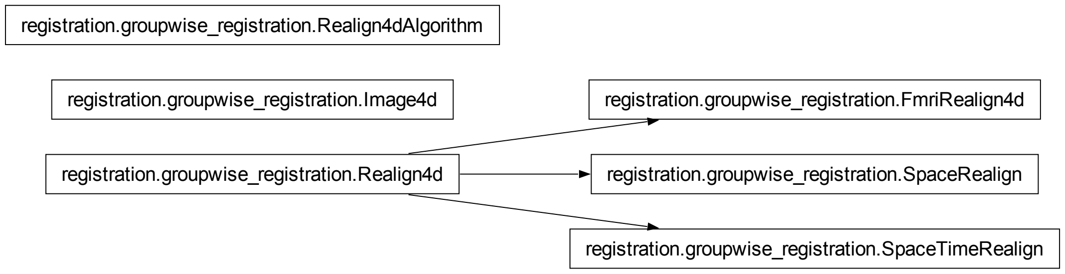
CHAPTER

FORTYTWO

ALGORITHMS.REGISTRATION.GROUPWISE_REGISTRATION

42.1 Module: `algorithms.registration.groupwise_registration`

Inheritance diagram for `nipy.algorithms.registration.groupwise_registration`:


registration.groupwise_registration.Realign4dAlgorithm
registration.groupwise_registration.Image4d
registration.groupwise_registration.Realign4d
registration.groupwise_registration.FmriRealign4d
registration.groupwise_registration.SpaceRealign
registration.groupwise_registration.SpaceTimeRealign

Motion correction / motion correction with slice timing

Routines implementing motion correction and motion correction combined with slice-timing.

See:

Roche, Alexis (2011) A four-dimensional registration algorithm with application to joint correction of motion and slice timing in fMRI. *Medical Imaging, IEEE Transactions on*; 30:1546–1554

42.2 Classes

42.2.1 `FmriRealign4d`

class nipy.algorithms.registration.groupwise_registration.**FmriRealign4d**(*images*, *slice_order=None*, *tr=None*, *tr_slices=None*, *start=0.0*, *interleaved=None*, *time_interp=None*, *slice_times=None*, *affine_class=<class 'nipy.algorithms.registration.affine.Rigid*, *slice_info=None*)

Bases: `Realign4d`

`__init__`(*images*, *slice_order=None*, *tr=None*, *tr_slices=None*, *start=0.0*, *interleaved=None*, *time_interp=None*, *slice_times=None*, *affine_class=<class 'nipy.algorithms.registration.affine.Rigid'>*, *slice_info=None*)

Spatiotemporal realignment class for fMRI series. This class is similar to *Realign4d* but provides a more flexible API for initialization in order to make it easier to declare slice acquisition times for standard sequences.

Warning: this class is deprecated; please use `SpaceTimeRealign` instead.

Parameters

images
[image or list of images] Single or multiple input 4d images representing one or several fMRI runs.

slice_order
[str or array-like] If str, one of {'ascending', 'descending'}. If array-like, then the order in which the slices were collected in time. For instance, the following represents an ascending contiguous sequence:

slice_order = [0, 1, 2, ...]

Note that *slice_order* differs from the argument used e.g. in the SPM slice timing routine in that it maps spatial slice positions to slice times. It is a mapping from space to time, while SPM conventionally uses the reverse mapping from time to space. For example, for an interleaved sequence with 10 slices, where we acquired slice 0 (in space) first, then slice 2 (in space) etc, *slice_order* would be [0, 5, 1, 6, 2, 7, 3, 8, 4, 9]

Using *slice_order* assumes that the inter-slice acquisition time is constant throughout acquisition. If this is not the case, use the *slice_times* argument instead and leave *slice_order* to None.

tr
[float] Inter-scan repetition time, i.e. the time elapsed between two consecutive scans. The unit in which *tr* is given is arbitrary although it needs to be consistent with the *tr_slices* and *start* arguments if provided. If None, *tr* is computed internally assuming a regular slice acquisition scheme.

tr_slices
[float] Inter-slice repetition time, same as *tr* for slices. If None, acquisition is assumed regular and *tr_slices* is set to *tr* divided by the number of slices.

start
[float] Starting acquisition time (time of the first acquired slice) respective to the time origin for resampling. *start* is assumed to be given in the same unit as *tr*. Setting *start=0* means that the resampled data will be synchronous with the first acquired slice. Setting *start=-tr/2*

means that the resampled data will be synchronous with the slice acquired at half repetition time.

time_interp: bool
Tells whether time interpolation is used or not within the realignment algorithm. If False, slices are considered to be acquired all at the same time, thus no slice timing correction will be performed.

interleaved
[bool] Deprecated argument.

Tells whether slice acquisition order is interleaved in a certain sense. Setting *interleaved* to True or False will trigger an error unless *slice_order* is 'ascending' or 'descending' and *slice_times* is None.

If slice_order=='ascending' and interleaved==True, the assumed slice order is (assuming 10 slices):

[0, 5, 1, 6, 2, 7, 3, 8, 4, 9]

If slice_order=='descending' and interleaved==True, the assumed slice order is:

[9, 4, 8, 3, 7, 2, 6, 1, 5, 0]

WARNING: given that there exist other types of interleaved acquisitions depending on scanner settings and manufacturers, you should refrain from using the *interleaved* keyword argument unless you are sure what you are doing. It is generally safer to explicitly input *slice_order* or *slice_times*.

slice_times
[None, str or array-like] This argument can be used instead of *slice_order*, *tr_slices*, *start* and *time_interp* altogether.

If None, slices are assumed to be acquired simultaneously hence no slice timing correction is performed. If array-like, then *slice_times* gives the slice acquisition times along the slice axis in units that are consistent with the provided *tr*.

Generally speaking, the following holds for sequences with constant inter-slice repetition time *tr_slices*:

slice_times = *start* + *tr_slices* * *slice_order*

For other sequences such as, e.g., sequences with simultaneously acquired slices, it is necessary to input *slice_times* explicitly along with *tr*.

slice_info
[None or tuple, optional] None, or a tuple with slice axis as the first element and direction as the second, for instance (2, 1). If None, then the slice axis and direction are guessed from the first run's affine assuming that slices are collected along the closest axis to the z-axis. This means that we assume by default an axial acquisition with slice axis pointing from bottom to top of the head.

`estimate`(*loops=5*, *between_loops=None*, *align_runs=True*, *speedup=5*, *refscan=0*, *borders=(1, 1, 1)*, *optimizer='ncg'*, *xtol=1e-05*, *ftol=1e-05*, *gtol=1e-05*, *stepsize=1e-06*, *maxiter=64*, *maxfun=None*)

Estimate motion parameters.

Parameters

loops
[int or sequence of ints] Determines the number of iterations performed to realign scans within each run for each pass defined by the `speedup` argument. For instance, setting `speedup` == (5,2) and `loops` == (5,1) means that 5 iterations are performed in a first pass

where scans are subsampled by an isotropic factor 5, followed by one iteration where scans are subsampled by a factor 2.

between_loops
[None, int or sequence of ints] Similar to `loops` for between-run motion estimation. Determines the number of iterations used to realign scans across runs, a procedure similar to within-run realignment that uses the mean images from each run. If None, assumed to be the same as `loops`. The setting used in the experiments described in Roche, IEEE TMI 2011, was: `speedup` = (5, 2), `loops` = (5, 1) and `between_loops` = (5, 1).

align_runs
[bool] Determines whether between-run motion is estimated or not. If False, the `between_loops` argument is ignored.

speedup: int or sequence of ints
Determines an isotropic sub-sampling factor, or a sequence of such factors, applied to the scans to perform motion estimation. If a sequence, several estimation passes are applied.

refscan
[None or int] Defines the number of the scan used as the reference coordinate system for each run. If None, a reference coordinate system is defined internally that does not correspond to any particular scan. Note that the coordinate system associated with the first run is always

borders
[sequence of ints] Should be of length 3. Determines the field of view for motion estimation in terms of the number of slices at each extremity of the reference grid that are ignored for motion parameter estimation. For instance, `` borders``==(1,1,1) means that the realignment cost function will not take into account voxels located in the first and last axial/sagittal/coronal slices in the reference grid. Please note that this choice only affects parameter estimation but does not affect image resampling in any way, see ``resample `` method.

optimizer
[str] Defines the optimization method. One of 'simplex', 'powell', 'cg', 'ncg', 'bfgs' and 'steepest'.

xtol
[float] Tolerance on variations of transformation parameters to test numerical convergence.

ftol
[float] Tolerance on variations of the intensity comparison metric to test numerical convergence.

gtol
[float] Tolerance on the gradient of the intensity comparison metric to test numerical convergence. Applicable to optimizers 'cg', 'ncg', 'bfgs' and 'steepest'.

stepsize
[float] Step size to approximate the gradient and Hessian of the intensity comparison metric w.r.t. transformation parameters. Applicable to optimizers 'cg', 'ncg', 'bfgs' and 'steepest'.

maxiter
[int] Maximum number of iterations in optimization.

maxfun
[int] Maximum number of function evaluations in maxfun.

resample(*r=None*, *align_runs=True*)

Return the resampled run number r as a 4d nipy-like image. Returns all runs as a list of images if r is None.

42.2.2 Image4d

class nipy.algorithms.registration.groupwise_registration.**Image4d**(*data*, *affine*, *tr*, *slice_times*, *slice_info=None*)

Bases: `object`

Class to represent a sequence of 3d scans (possibly acquired on a slice-by-slice basis).

Object remains empty until the data array is actually loaded in memory.

Parameters

data
[nd array or proxy (function that actually gets the array)]

__init__(*data*, *affine*, *tr*, *slice_times*, *slice_info=None*)

Configure fMRI acquisition time parameters.

free_data()

get_fdata()

get_shape()

scanner_time(*zv*, *t*)

tv = scanner_time(zv, t) zv, tv are grid coordinates; t is an actual time value.

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)

42.2.3 Realign4d

class nipy.algorithms.registration.groupwise_registration.**Realign4d**(*images*, *tr*, *slice_times=None*, *slice_info=None*, *affine_class=<class 'nipy.algorithms.registration.affine.Rigid'>*)

Bases: `object`

__init__(*images*, *tr*, *slice_times=None*, *slice_info=None*, *affine_class=<class 'nipy.algorithms.registration.affine.Rigid'>*)

Spatiotemporal realignment class for series of 3D images.

The algorithm performs simultaneous motion and slice timing correction for fMRI series or other data where slices are not acquired simultaneously.

Parameters

images
[image or list of images] Single or multiple input 4d images representing one or several sessions.

tr
[float] Inter-scan repetition time, i.e. the time elapsed between two consecutive scans. The unit in which *tr* is given is arbitrary although it needs to be consistent with the *slice_times* argument.

slice_times
[None or array-like] If None, slices are assumed to be acquired simultaneously hence no slice timing correction is performed. If array-like, then the slice acquisition times.

slice_info
[None or tuple, optional] None, or a tuple with slice axis as the first element and direction as the second, for instance (2, 1). If None, then guess the slice axis, and direction, as the closest to the z axis, as estimated from the affine.

`estimate`(*loops=5*, *between_loops=None*, *align_runs=True*, *speedup=5*, *refscan=0*, *borders=(1, 1, 1)*, *optimizer='ncg'*, *xtol=1e-05*, *ftol=1e-05*, *gtol=1e-05*, *stepsize=1e-06*, *maxiter=64*, *maxfun=None*)

Estimate motion parameters.

Parameters

loops
[int or sequence of ints] Determines the number of iterations performed to realign scans within each run for each pass defined by the `speedup` argument. For instance, setting `speedup` == (5,2) and `loops` == (5,1) means that 5 iterations are performed in a first pass where scans are subsampled by an isotropic factor 5, followed by one iteration where scans are subsampled by a factor 2.

between_loops
[None, int or sequence of ints] Similar to `loops` for between-run motion estimation. Determines the number of iterations used to realign scans across runs, a procedure similar to within-run realignment that uses the mean images from each run. If None, assumed to be the same as `loops`. The setting used in the experiments described in Roche, IEEE TMI 2011, was: `speedup` = (5, 2), `loops` = (5, 1) and `between_loops` = (5, 1).

align_runs
[bool] Determines whether between-run motion is estimated or not. If False, the `between_loops` argument is ignored.

speedup: int or sequence of ints
Determines an isotropic sub-sampling factor, or a sequence of such factors, applied to the scans to perform motion estimation. If a sequence, several estimation passes are applied.

refscan
[None or int] Defines the number of the scan used as the reference coordinate system for each run. If None, a reference coordinate system is defined internally that does not correspond to any particular scan. Note that the coordinate system associated with the first run is always

borders
[sequence of ints] Should be of length 3. Determines the field of view for motion estimation in terms of the number of slices at each extremity of the reference grid that are ignored for motion parameter estimation. For instance, `borders``==(1,1,1) means that the realignment cost function will not take into account voxels located in the first and last axial/sagittal/coronal slices in the reference grid. Please note that this choice only affects parameter estimation but does not affect image resampling in any way, see ``resample` method.

optimizer
[str] Defines the optimization method. One of 'simplex', 'powell', 'cg', 'ncg', 'bfgs' and 'steepest'.

xtol
[float] Tolerance on variations of transformation parameters to test numerical convergence.

ftol
[float] Tolerance on variations of the intensity comparison metric to test numerical convergence.

gtol
[float] Tolerance on the gradient of the intensity comparison metric to test numerical convergence. Applicable to optimizers 'cg', 'ncg', 'bfgs' and 'steepest'.

stepsize
[float] Step size to approximate the gradient and Hessian of the intensity comparison metric w.r.t. transformation parameters. Applicable to optimizers 'cg', 'ncg', 'bfgs' and 'steepest'.

maxiter
[int] Maximum number of iterations in optimization.

maxfun
[int] Maximum number of function evaluations in maxfun.

resample(*r=None*, *align_runs=True*)

Return the resampled run number r as a 4d nipy-like image. Returns all runs as a list of images if r is None.

42.2.4 Realign4dAlgorithm

class nipy.algorithms.registration.groupwise_registration.**Realign4dAlgorithm**(*im4d*, *affine_class=<class 'nipy.algorithms.registration.affine*, *transforms=None*, *time_interp=True*, *subsampling=(1, 1, 1)*, *refscan=0*, *borders=(1, 1, 1)*, *optimizer='ncg'*, *optimize_template=True*, *xtol=1e-05*, *ftol=1e-05*, *gtol=1e-05*, *stepsize=1e-06*, *maxiter=64*, *maxfun=None*)

Bases: object

__init__(*im4d*, *affine_class=<class 'nipy.algorithms.registration.affine.Rigid'>*, *transforms=None*, *time_interp=True*, *subsampling=(1, 1, 1)*, *refscan=0*, *borders=(1, 1, 1)*, *optimizer='ncg'*, *optimize_template=True*, *xtol=1e-05*, *ftol=1e-05*, *gtol=1e-05*, *stepsize=1e-06*, *maxiter=64*, *maxfun=None*) `align_to_refscan()`

The *motion_estimate* method aligns scans with an online template so that spatial transforms map some average head space to the scanner space. To conventionally redefine the head space as being aligned with some reference scan, we need to right compose each head_average-to-scanner transform with the refscan's 'to head_average' transform.

`estimate_instant_motion(`*t*`)`

Estimate motion parameters at a particular time.

`estimate_motion()`

Optimize motion parameters for the whole sequence. All the time frames are initially resampled according to the current space/time transformation, the parameters of which are further optimized sequentially.

`init_instant_motion(`*t*`)`

Pre-compute and cache some constants (at fixed time) for repeated computations of the alignment energy.

The idea is to decompose the average temporal variance via:

V = (n-1)/n V* + (n-1)/n^2 (x-m*)^2

with x the considered volume at time t, and m* the mean of all resampled volumes but x. Only the second term is variable when

one volumes while the others are fixed. A similar decomposition is used for the global variance, so we end up with:

V/V0 = [nV* + (x-m*)^2] / [nV0* + (x-m0*)^2]

`resample(`*t*`)`

Resample a particular time frame on the (sub-sampled) working grid.

x,y,z,t are "head" grid coordinates X,Y,Z,T are "scanner" grid coordinates

`resample_full_data()`

`set_fmin(`*optimizer*, *stepsize*, ***kwargs*`)`

Return the minimization function

`set_transform(`*t*, *pc*`)`

42.2.5 SpaceRealign

`class nipy.algorithms.registration.groupwise_registration.SpaceRealign(`*images*, *affine_class=<class 'nipy.algorithms.registration.affine.Rigid'>*

Bases: `Realign4d`

`__init__(`*images*, *affine_class=<class 'nipy.algorithms.registration.affine.Rigid'>*`)`

Spatial registration of time series with no time interpolation

Parameters

images
[image or list of images] Single or multiple input 4d images representing one or several fMRI runs.

affine_class
[`Affine` class, optional] transformation class to use to calculate transformations between the volumes. Default is :class:`Rigid`

`estimate`(*loops=5*, *between_loops=None*, *align_runs=True*, *speedup=5*, *refscan=0*, *borders=(1, 1, 1)*, *optimizer='ncg'*, *xtol=1e-05*, *ftol=1e-05*, *gtol=1e-05*, *stepsize=1e-06*, *maxiter=64*, *maxfun=None*)

Estimate motion parameters.

Parameters

loops
[int or sequence of ints] Determines the number of iterations performed to realign scans within each run for each pass defined by the `speedup` argument. For instance, setting `speedup` == (5,2) and `loops` == (5,1) means that 5 iterations are performed in a first pass where scans are subsampled by an isotropic factor 5, followed by one iteration where scans are subsampled by a factor 2.

between_loops
[None, int or sequence of ints] Similar to `loops` for between-run motion estimation. Determines the number of iterations used to realign scans across runs, a procedure similar to within-run realignment that uses the mean images from each run. If None, assumed to be the same as `loops`. The setting used in the experiments described in Roche, IEEE TMI 2011, was: `speedup` = (5, 2), `loops` = (5, 1) and `between_loops` = (5, 1).

align_runs
[bool] Determines whether between-run motion is estimated or not. If False, the `between_loops` argument is ignored.

speedup: int or sequence of ints
Determines an isotropic sub-sampling factor, or a sequence of such factors, applied to the scans to perform motion estimation. If a sequence, several estimation passes are applied.

refscan
[None or int] Defines the number of the scan used as the reference coordinate system for each run. If None, a reference coordinate system is defined internally that does not correspond to any particular scan. Note that the coordinate system associated with the first run is always

borders
[sequence of ints] Should be of length 3. Determines the field of view for motion estimation in terms of the number of slices at each extremity of the reference grid that are ignored for motion parameter estimation. For instance, `` borders``==(1,1,1) means that the realignment cost function will not take into account voxels located in the first and last axial/sagittal/coronal slices in the reference grid. Please note that this choice only affects parameter estimation but does not affect image resampling in any way, see ``resample `` method.

optimizer
[str] Defines the optimization method. One of 'simplex', 'powell', 'cg', 'ncg', 'bfgs' and 'steepest'.

xtol
[float] Tolerance on variations of transformation parameters to test numerical convergence.

ftol
[float] Tolerance on variations of the intensity comparison metric to test numerical convergence.

gtol
[float] Tolerance on the gradient of the intensity comparison metric to test numerical convergence. Applicable to optimizers 'cg', 'ncg', 'bfgs' and 'steepest'.

stepsize
[float] Step size to approximate the gradient and Hessian of the intensity comparison metric w.r.t. transformation parameters. Applicable to optimizers 'cg', 'ncg', 'bfgs' and 'steepest'.

maxiter
[int] Maximum number of iterations in optimization.

maxfun
[int] Maximum number of function evaluations in maxfun.

resample(*r=None*, *align_runs=True*)

Return the resampled run number r as a 4d nipy-like image. Returns all runs as a list of images if r is None.

42.2.6 SpaceTimeRealign

class nipy.algorithms.registration.groupwise_registration.**SpaceTimeRealign**(*images*, *tr*, *slice_times*, *slice_info*, *affine_class=<class 'nipy.algorithms.registration.affine.R*

Bases: `Realign4d`

__init__(*images*, *tr*, *slice_times*, *slice_info*, *affine_class=<class 'nipy.algorithms.registration.affine.Rigid'>*)

Spatiotemporal realignment class for fMRI series.

This class gives a high-level interface to `Realign4d`

Parameters

images
[image or list of images] Single or multiple input 4d images representing one or several fMRI runs.

tr
[None or float or "header-allow-1.0"] Inter-scan repetition time in seconds, i.e. the time elapsed between two consecutive scans. If None, an attempt is made to read the TR from the header, but an exception is thrown for values 0 or 1. A value of "header-allow-1.0" will signal to accept a header TR of 1.

slice_times
[str or callable or array-like] If str, one of the function names in `SLICETIME_FUNCTIONS` dictionary from `nipy.algorithms.slicetiming.timefuncs`. If callable, a function taking two parameters: `n_slices` and `tr` (number of slices in the images, inter-scan repetition time in seconds). This function returns a vector of times of slice acquisition t_i for each slice i in the volumes. See `nipy.algorithms.slicetiming.timefuncs` for a collection of functions for common slice acquisition schemes. If array-like, then should be a slice time vector as above.

slice_info
[int or length 2 sequence] If int, the axis in *images* that is the slice axis. In a 4D image, this will often be axis = 2. If a 2 sequence, then elements are `(slice_axis, slice_direction)`, where `slice_axis` is the slice axis in the image as above, and `slice_direction` is 1 if the slices were acquired slice 0 first, slice -1 last, or -1 if acquired slice -1 first, slice 0 last. If *slice_info* is an int, assume `slice_direction` == 1.

affine_class
[`Affine` class, optional] transformation class to use to calculate transformations between the volumes. Default is :class:`Rigid`

`estimate`(*loops=5*, *between_loops=None*, *align_runs=True*, *speedup=5*, *refscan=0*, *borders=(1, 1, 1)*, *optimizer='ncg'*, *xtol=1e-05*, *ftol=1e-05*, *gtol=1e-05*, *stepsize=1e-06*, *maxiter=64*, *maxfun=None*)

Estimate motion parameters.

Parameters

loops
[int or sequence of ints] Determines the number of iterations performed to realign scans within each run for each pass defined by the `speedup` argument. For instance, setting `speedup` == (5,2) and `loops` == (5,1) means that 5 iterations are performed in a first pass where scans are subsampled by an isotropic factor 5, followed by one iteration where scans are subsampled by a factor 2.

between_loops
[None, int or sequence of ints] Similar to `loops` for between-run motion estimation. Determines the number of iterations used to realign scans across runs, a procedure similar to within-run realignment that uses the mean images from each run. If None, assumed to be the same as `loops`. The setting used in the experiments described in Roche, IEEE TMI 2011, was: `speedup` = (5, 2), `loops` = (5, 1) and `between_loops` = (5, 1).

align_runs
[bool] Determines whether between-run motion is estimated or not. If False, the `between_loops` argument is ignored.

speedup: int or sequence of ints
Determines an isotropic sub-sampling factor, or a sequence of such factors, applied to the scans to perform motion estimation. If a sequence, several estimation passes are applied.

refscan
[None or int] Defines the number of the scan used as the reference coordinate system for each run. If None, a reference coordinate system is defined internally that does not correspond to any particular scan. Note that the coordinate system associated with the first run is always

borders
[sequence of ints] Should be of length 3. Determines the field of view for motion estimation in terms of the number of slices at each extremity of the reference grid that are ignored for motion parameter estimation. For instance, `` borders``==(1,1,1) means that the realignment cost function will not take into account voxels located in the first and last axial/sagittal/coronal slices in the reference grid. Please note that this choice only affects parameter estimation but does not affect image resampling in any way, see ``resample `` method.

optimizer
[str] Defines the optimization method. One of 'simplex', 'powell', 'cg', 'ncg', 'bfgs' and 'steepest'.

xtol
[float] Tolerance on variations of transformation parameters to test numerical convergence.

ftol
[float] Tolerance on variations of the intensity comparison metric to test numerical convergence.

gtol
[float] Tolerance on the gradient of the intensity comparison metric to test numerical convergence. Applicable to optimizers 'cg', 'ncg', 'bfgs' and 'steepest'.

stepsize
[float] Step size to approximate the gradient and Hessian of the intensity comparison metric w.r.t. transformation parameters. Applicable to optimizers 'cg', 'ncg', 'bfgs' and 'steepest'.

maxiter
[int] Maximum number of iterations in optimization.

maxfun
[int] Maximum number of function evaluations in maxfun.

resample(*r=None*, *align_runs=True*)
Return the resampled run number r as a 4d nipy-like image. Returns all runs as a list of images if r is None.

42.3 Functions

`nipy.algorithms.registration.groupwise_registration.`**adjust_subsampling**(*speedup*, *dims*)

`nipy.algorithms.registration.groupwise_registration.`**guess_slice_axis_and_direction**(*slice_info*, *affine*)

`nipy.algorithms.registration.groupwise_registration.`**interp_slice_times**(*Z*, *slice_times*, *tr*)

`nipy.algorithms.registration.groupwise_registration.`**make_grid**(*dims*, *subsampling=(1, 1, 1)*, *borders=(0, 0, 0)*)

`nipy.algorithms.registration.groupwise_registration.`**realign4d**(*runs*, *affine_class=<class 'nipy.algorithms.registration.affine.Rigid'>*, *time_interp=True*, *align_runs=True*, *loops=5*, *between_loops=5*, *speedup=5*, *refscan=0*, *borders=(1, 1, 1)*, *optimizer='ncg'*, *xtol=1e-05*, *ftol=1e-05*, *gtol=1e-05*, *stepsize=1e-06*, *maxiter=64*, *maxfun=None*)

Parameters

runs
[list of Image4d objects]

Returns

transforms
[list] nested list of rigid transformations

transforms map an 'ideal' 4d grid (conventionally aligned with the first scan of the first run) to the 'acquisition' 4d grid for each run

`nipy.algorithms.registration.groupwise_registration.`**resample4d**(*im4d*, *transforms*, *time_interp=True*)

Resample a 4D image according to the specified sequence of spatial transforms, using either 4D interpolation if *time_interp* is True and 3D interpolation otherwise.

`nipy.algorithms.registration.groupwise_registration.`**`scanner_coords`**(*xyz*, *affine*, *from_world*, *to_world*)

`nipy.algorithms.registration.groupwise_registration.`**`single_run_realign4d`**(*im4d*, *affine_class=<class 'nipy.algorithms.registration.affine.Rigi*, *time_interp=True*, *loops=5*, *speedup=5*, *refscan=0*, *borders=(1, 1, 1)*, *optimizer='ncg'*, *xtol=1e-05*, *ftol=1e-05*, *gtol=1e-05*, *stepsize=1e-06*, *maxiter=64*, *maxfun=None*)

Realign a single run in space and time.

Parameters

im4d
[Image4d instance]

speedup
[int or sequence] If a sequence, implement a multi-scale realignment

`nipy.algorithms.registration.groupwise_registration.`**`tr_from_header`**(*images*)

Return the TR from the header of an image or list of images.

Parameters

images
[image or list of images] Single or multiple input 4d images representing one or several sessions.

Returns

float
Repetition time, as specified in NIfTI header.

Raises

ValueError
if the TR between the images is inconsistent.

CHAPTER

FORTYTHREE

ALGORITHMS.REGISTRATION.HISTOGRAM_REGISTRATION

43.1 Module: `algorithms.registration.histogram_registration`

Inheritance diagram for `nipy.algorithms.registration.histogram_registration`:

Intensity-based image registration

43.2 Class

43.3 `HistogramRegistration`

class `nipy.algorithms.registration.histogram_registration.`**`HistogramRegistration`**(*from_img*, *to_img*, *from_bins=256*, *to_bins=None*, *from_mask=None*, *to_mask=None*, *similarity='crl1'*, *interp='pv'*, *smooth=0*, *renormalize=False*, *dist=None*)

Bases: `object`

A class to represent a generic intensity-based image registration algorithm.

`__init__`(*from_img*, *to_img*, *from_bins=256*, *to_bins=None*, *from_mask=None*, *to_mask=None*, *similarity='crl1'*, *interp='pv'*, *smooth=0*, *renormalize=False*, *dist=None*)

Creates a new histogram registration object.

Parameters

from_img
[nipy-like image]

From image

to_img
[nipy-like image] *To* image

from_bins
[integer] Number of histogram bins to represent the *from* image

to_bins
[integer] Number of histogram bins to represent the *to* image

from_mask
[array-like] Mask to apply to the *from* image

to_mask
[array-like] Mask to apply to the *to* image

similarity
[str or callable] Cost-function for assessing image similarity. If a string, one of 'cc': correlation coefficient, 'cr': correlation ratio, 'crl1': L1-norm based correlation ratio, 'mi': mutual information, 'nmi': normalized mutual information, 'slr': supervised log-likelihood ratio. If a callable, it should take a two-dimensional array representing the image joint histogram as an input and return a float.

dist: None or array-like
Joint intensity probability distribution model for use with the 'slr' measure. Should be of shape (from_bins, to_bins).

interp
[str] Interpolation method. One of 'pv': Partial volume, 'tri': Trilinear, 'rand': Random interpolation. See `joint_histogram.c`

smooth
[float] Standard deviation in millimeters of an isotropic Gaussian kernel used to smooth the *To* image. If 0, no smoothing is applied.

`eval`(*T*)

Evaluate similarity function given a world-to-world transform.

Parameters

T
[Transform] Transform object implementing `apply` method

`eval_gradient`(*T*, *epsilon=0.1*)

Evaluate the gradient of the similarity function wrt transformation parameters.

The gradient is approximated using central finite differences at the transformation specified by *T*. The input transformation object *T* is modified in place unless it has a `copy` method.

Parameters

T
[Transform] Transform object implementing `apply` method

epsilon
[float] Step size for finite differences in units of the transformation parameters

Returns

g
[ndarray] Similarity gradient estimate

`eval_hessian`(*T*, *epsilon=0.1*, *diag=False*)

Evaluate the Hessian of the similarity function wrt transformation parameters.

The Hessian or its diagonal is approximated at the transformation specified by *T* using central finite differences. The input transformation object *T* is modified in place unless it has a `copy` method.

Parameters

T
[Transform] Transform object implementing `apply` method

epsilon
[float] Step size for finite differences in units of the transformation parameters

diag
[bool] If True, approximate the Hessian by a diagonal matrix.

Returns

H
[ndarray] Similarity Hessian matrix estimate

`explore`(*T*, **args*)

Evaluate the similarity at the transformations specified by sequences of parameter values.

For instance:

s, p = explore(T, (0, [-1,0,1]), (4, [-2.,2]))

Parameters

T
[object] Transformation around which the similarity function is to be evaluated. It is modified in place unless it has a `copy` method.

args
[tuple] Each element of *args* is a sequence of two elements, where the first element specifies a transformation parameter axis and the second element gives the successive parameter values to evaluate along that axis.

Returns

s
[ndarray] Array of similarity values

p
[ndarray] Corresponding array of evaluated transformation parameters

`property interp`

`optimize`(*T*, *optimizer='powell'*, ***kwargs*)

Optimize transform *T* with respect to similarity measure.

The input object *T* will change as a result of the optimization.

Parameters

T
[object or str] An object representing a transformation that should implement `apply` method and `param` attribute or property. If a string, one of 'rigid', 'similarity', or 'affine'. The corresponding transformation class is then initialized by default.

optimizer
[str] Name of optimization function (one of 'powell', 'steepest', 'cg', 'bfgs', 'simplex')

****kwargs**
[dict] keyword arguments to pass to optimizer

Returns

T
[object] Locally optimal transformation

`set_fov`(*spacing=None*, *corner=(0, 0, 0)*, *size=None*, *npoints=None*)

Defines a subset of the *from* image to restrict joint histogram computation.

Parameters

spacing
[sequence (3,) of positive integers] Subsampling of image in voxels, where None (default) results in the subsampling to be automatically adjusted to roughly match a cubic grid with *npoints* voxels

corner
[sequence (3,) of positive integers] Bounding box origin in voxel coordinates

size
[sequence (3,) of positive integers] Desired bounding box size

npoints
[positive integer] Desired number of voxels in the bounding box. If a *spacing* argument is provided, then *npoints* is ignored.

`property similarity`

`subsample`(*spacing=None*, *npoints=None*)

43.4 Functions

`nipy.algorithms.registration.histogram_registration.approx_gradient`(*f*, *x*, *epsilon*)

Approximate the gradient of a function using central finite differences

Parameters

f: callable
The function to differentiate

x: ndarray
Point where the function gradient is to be evaluated

epsilon: float
Stepsize for finite differences

Returns

g: ndarray
Function gradient at *x*

`nipy.algorithms.registration.histogram_registration.`**`approx_hessian`**`(`*f*, *x*, *epsilon*`)`

Approximate the full Hessian matrix of a function using central finite differences

Parameters

f: callable
The function to differentiate

x: ndarray
Point where the Hessian is to be evaluated

epsilon: float
Stepsize for finite differences

Returns

H: ndarray
Hessian matrix at *x*

`nipy.algorithms.registration.histogram_registration.`**`approx_hessian_diag`**`(`*f*, *x*, *epsilon*`)`

Approximate the Hessian diagonal of a function using central finite differences

Parameters

f: callable
The function to differentiate

x: ndarray
Point where the Hessian is to be evaluated

epsilon: float
Stepsize for finite differences

Returns

h: ndarray
Diagonal of the Hessian at *x*

`nipy.algorithms.registration.histogram_registration.`**`clamp`**`(`*x*, *bins*, *mask=None*`)`

Clamp array values that fall within a given mask in the range [0..bins-1] and reset masked values to -1.

Parameters

x
[ndarray] The input array

bins
[number] Desired number of bins

mask
[ndarray, tuple or slice] Anything such that x[mask] is an array.

Returns

y
[ndarray] Clamped array, masked items are assigned -1

bins
[number] Adjusted number of bins

`nipy.algorithms.registration.histogram_registration.`**`ideal_spacing`**`(`*data*, *npoints*`)`

Tune spacing factors so that the number of voxels in the output block matches a given number.

Parameters

data
[ndarray or sequence] Data image to subsample

npoints
[number] Target number of voxels (negative values will be ignored)

Returns

spacing: ndarray
Spacing factors

`nipy.algorithms.registration.histogram_registration.`**`smallest_bounding_box`**(*msk*)

Extract the smallest bounding box from a mask

Parameters

msk
[ndarray] Array of boolean

Returns

corner: ndarray
3-dimensional coordinates of bounding box corner

size: ndarray
3-dimensional size of bounding box

`nipy.algorithms.registration.histogram_registration.`**`smooth_image`**(*data*, *affine*, *sigma*)

Smooth an image by an isotropic Gaussian filter

Parameters

data: ndarray
Image data array

affine: ndarray
Image affine transform

sigma: float
Filter standard deviation in mm

Returns

sdata: ndarray
Smoothed data array

CHAPTER

FORTYFOUR

ALGORITHMS.REGISTRATION.OPTIMIZER

44.1 Module: `algorithms.registration.optimizer`

44.2 Functions

`nipy.algorithms.registration.optimizer.`**`configure_optimizer`**(*optimizer*, *fprime=None*, *fhess=None*, ***kwargs*)

Return the minimization function

`nipy.algorithms.registration.optimizer.`**`subdict`**(*dic*, *keys*)

`nipy.algorithms.registration.optimizer.`**`use_derivatives`**(*optimizer*)

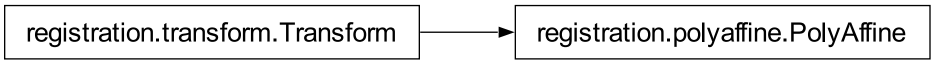
CHAPTER

FORTYFIVE

ALGORITHMS.REGISTRATION.POLYAFFINE

45.1 Module: `algorithms.registration.polyaffine`

Inheritance diagram for `nipy.algorithms.registration.polyaffine`:


registration.transform.Transform
registration.polyaffine.PolyAffine

45.2 `PolyAffine`

class `nipy.algorithms.registration.polyaffine.`**`PolyAffine`**(*centers*, *affines*, *sigma*, *glob_affine=None*)

Bases: `Transform`

`__init__`(*centers*, *affines*, *sigma*, *glob_affine=None*)

centers: N times 3 array

We are given a set of affine transforms T_i with centers x_i, all in homogeneous coordinates. The polyaffine transform is defined, up to a right composition with a global affine, as:

T(x) = sum_i w_i(x) T_i x

where w_i(x) = g(x-x_i)/Z(x) are normalized Gaussian weights that sum up to one for every x.

`affine`(*i*)

`affines`()

`apply`(*xyz*)

xyz is an (N, 3) array

`compose`(*other*)

Compose this transform onto another

Parameters

other

[Transform] transform that we compose onto

Returns

composed_transform
[Transform] a transform implementing the composition of self on *other*

`left_compose`(*other*)

`property param`

CHAPTER

FORTYSIX

ALGORITHMS.REGISTRATION.RESAMPLE

46.1 Module: `algorithms.registration.resample`

46.2 Functions

`nipy.algorithms.registration.resample.`**`cast_array`**(*arr*, *dtype*)

arr
[array] Input array

dtype
[dtype] Desired dtype

`nipy.algorithms.registration.resample.`**`resample`**(*moving*, *transform=None*, *reference=None*, *mov_voxel_coords=False*, *ref_voxel_coords=False*, *dtype=None*, *interp_order=3*, *mode='constant'*, *cval=0.0*)

Resample *movimg* into voxel space of *reference* using *transform*

Apply a transformation to the image considered as 'moving' to bring it into the same grid as a given *reference* image. The transformation usually maps world space in *reference* to world space in *movimg*, but can also be a voxel to voxel mapping (see parameters below).

This function uses scipy.ndimage except for the case *interp_order==3*, where a fast cubic spline implementation is used.

Parameters

moving: nipy-like image
Image to be resampled.

transform: transform object or None
Represents a transform that goes from the *reference* image to the *moving* image. None means an identity transform. Otherwise, it should have either an *apply* method, or an *as_affine* method or be a shape (4, 4) array. By default, *transform* maps between the output (world) space of *reference* and the output (world) space of *moving*. If *mov_voxel_coords* is True, maps to the *voxel* space of *moving* and if *ref_vox_coords* is True, maps from the *voxel* space of *reference*.

reference
[None or nipy-like image or tuple, optional] The reference image defines the image dimensions and xyz affine to which to resample. It can be input as a nipy-like image or as a tuple (shape, affine). If None, use *movimg* to define these.

mov_voxel_coords
[boolean, optional] True if the transform maps to voxel coordinates, False if it maps to world coordinates.

ref_voxel_coords
[boolean, optional] True if the transform maps from voxel coordinates, False if it maps from world coordinates.

interp_order: int, optional
Spline interpolation order, defaults to 3.

mode
[str, optional] Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect' or 'wrap'). Default is 'constant'.

cval
[scalar, optional] Value used for points outside the boundaries of the input if mode='constant'. Default is 0.0.

Returns

aligned_img
[Image] Image resliced to *reference* with reference-to-movimg transform *transform*

CHAPTER

FORTYSEVEN

ALGORITHMS.REGISTRATION.SCRIPTING

47.1 Module: `algorithms.registration.scripting`

A scripting wrapper around 4D registration (SpaceTimeRealign)

47.2 Functions

`nipy.algorithms.registration.scripting.`**`aff2euler`**(*affine*)

Compute Euler angles from 4 x 4 *affine*

Parameters

affine
[4 by 4 array] An affine transformation matrix

Returns

The Euler angles associated with the affine

`nipy.algorithms.registration.scripting.`**`aff2rot_zooms`**(*affine*)

Compute a rotation matrix and zooms from 4 x 4 *affine*

Parameters

affine
[4 by 4 array] An affine transformation matrix

Returns

R: 3 by 3 array
A rotation matrix in 3D

zooms: length 3 1-d array
Vector with voxel sizes.

`nipy.algorithms.registration.scripting.`**`space_time_realign`**(*input*, *tr*, *slice_order='descending'*, *slice_dim=2*, *slice_dir=1*, *apply=True*, *make_figure=False*, *out_name=None*)

This is a scripting interface to *nipy.algorithms.registration.SpaceTimeRealign*

Parameters

input
[str or list] A full path to a file-name (4D nifti time-series) , or to a directory containing 4D nifti time-series, or a list of full-paths to files.

tr
[float] The repetition time

slice_order
[str (optional)] This is the order of slice-times in the acquisition. This is used as a key into the `SLICETIME_FUNCTIONS` dictionary from `nipy.algorithms.slicetiming.timefuncs`. Default: 'descending'.

slice_dim
[int (optional)] Denotes the axis in *images* that is the slice axis. In a 4D image, this will often be axis = 2 (default).

slice_dir
[int (optional)] 1 if the slices were acquired slice 0 first (default), slice -1 last, or -1 if acquire slice -1 first, slice 0 last.

apply
[bool (optional)] Whether to apply the transformation and produce an output. Default: True.

make_figure
[bool (optional)] Whether to generate a .png figure with the parameters across scans.

out_name
[bool (optional)] Specify an output location (full path) for the files that are generated. Default: generate files in the path of the inputs (with an *_mc* suffix added to the file-names.

Returns

transforms
[ndarray]

An (n_times_points,) shaped array containing

***nipy.algorithms.registration.affine.Rigid* class instances for each time**
point in the time-series. These can be used as affine transforms by referring to their *.as_affine* attribute.

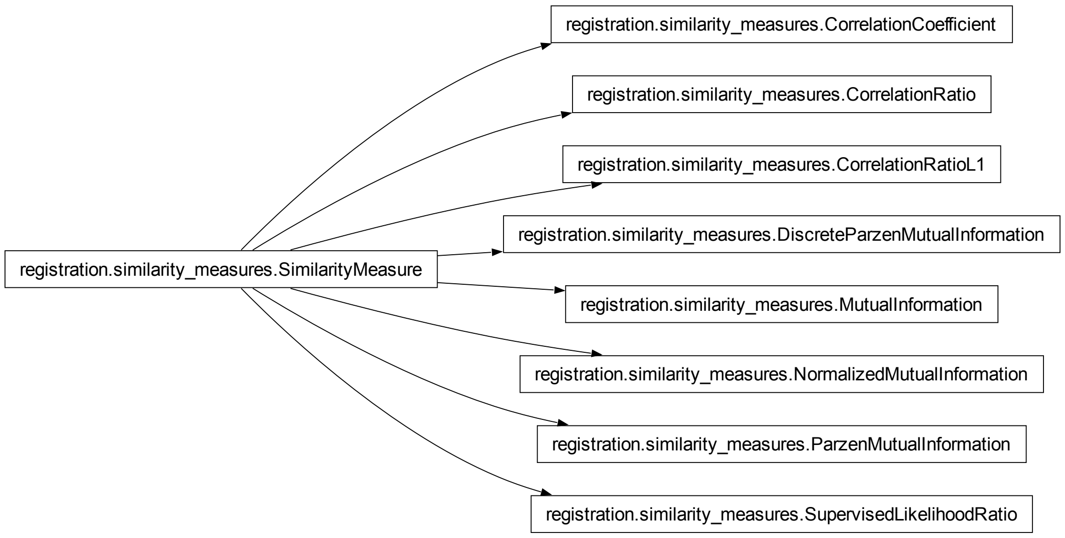
CHAPTER

FORTYEIGHT

ALGORITHMS.REGISTRATION.SIMILARITY_MEASURES

48.1 Module: `algorithms.registration.similarity_measures`

Inheritance diagram for `nipy.algorithms.registration.similarity_measures`:


registration.similarity_measures.SimilarityMeasure
registration.similarity_measures.CorrelationCoefficient
registration.similarity_measures.CorrelationRatio
registration.similarity_measures.CorrelationRatioL1
registration.similarity_measures.DiscreteParzenMutualInformation
registration.similarity_measures.MutualInformation
registration.similarity_measures.NormalizedMutualInformation
registration.similarity_measures.ParzenMutualInformation
registration.similarity_measures.SupervisedLikelihoodRatio

48.2 Classes

48.2.1 `CorrelationCoefficient`

class `nipy.algorithms.registration.similarity_measures.`**`CorrelationCoefficient`**(*shape*, *renormalize=False*, *dist=None*)

Bases: *`SimilarityMeasure`*

Use a bivariate Gaussian as a distribution model

`__init__`(*shape*, *renormalize=False*, *dist=None*)

`loss`(*H*)

`npoints`(*H*)

48.2.2 `CorrelationRatio`

class `nipy.algorithms.registration.similarity_measures.`**`CorrelationRatio`**(*shape*, *renormalize=False*, *dist=None*)

Bases: *`SimilarityMeasure`*

Use a nonlinear regression model with Gaussian errors as a distribution model

`__init__`(*shape*, *renormalize=False*, *dist=None*)

`loss`(*H*)

`npoints`(*H*)

48.2.3 `CorrelationRatioL1`

class `nipy.algorithms.registration.similarity_measures.`**`CorrelationRatioL1`**(*shape*, *renormalize=False*, *dist=None*)

Bases: *`SimilarityMeasure`*

Use a nonlinear regression model with Laplace distributed errors as a distribution model

`__init__`(*shape*, *renormalize=False*, *dist=None*)

`loss`(*H*)

`npoints`(*H*)

48.2.4 `DiscreteParzenMutualInformation`

class `nipy.algorithms.registration.similarity_measures.`**`DiscreteParzenMutualInformation`**(*shape*, *renormalize=False*, *dist=None*)

Bases: *`SimilarityMeasure`*

Use Parzen windowing in the discrete case to estimate the distribution model

`__init__`(*shape*, *renormalize=False*, *dist=None*)

`loss`(*H*)

`npoints`(*H*)

48.2.5 MutualInformation

class nipy.algorithms.registration.similarity_measures.**MutualInformation**(*shape*, *renormalize=False*, *dist=None*)

Bases: *SimilarityMeasure*

Use the normalized joint histogram as a distribution model

__init__(*shape*, *renormalize=False*, *dist=None*)

loss(*H*)

npoints(*H*)

48.2.6 NormalizedMutualInformation

class nipy.algorithms.registration.similarity_measures.**NormalizedMutualInformation**(*shape*, *renormalize=False*, *dist=None*)

Bases: *SimilarityMeasure*

NMI = 2*(1 - H(I,J)/[H(I)+H(J)])
= 2*MI/[H(I)+H(J)])

__init__(*shape*, *renormalize=False*, *dist=None*)

loss(*H*)

npoints(*H*)

48.2.7 ParzenMutualInformation

class nipy.algorithms.registration.similarity_measures.**ParzenMutualInformation**(*shape*, *renormalize=False*, *dist=None*)

Bases: *SimilarityMeasure*

Use Parzen windowing to estimate the distribution model

__init__(*shape*, *renormalize=False*, *dist=None*)

loss(*H*)

npoints(*H*)

48.2.8 SimilarityMeasure

class nipy.algorithms.registration.similarity_measures.**SimilarityMeasure**(*shape*, *renormalize=False*, *dist=None*)

Bases: object

Template class

__init__(*shape*, *renormalize=False*, *dist=None*)

loss(*H*)

npoints(*H*)

48.2.9 SupervisedLikelihoodRatio

class nipy.algorithms.registration.similarity_measures.**SupervisedLikelihoodRatio**(*shape*, *renormalize=False*, *dist=None*)

Bases: *SimilarityMeasure*

Assume a joint intensity distribution model is given by self.dist

__init__(*shape*, *renormalize=False*, *dist=None*)

loss(*H*)

npoints(*H*)

48.3 Functions

nipy.algorithms.registration.similarity_measures.**correlation2loglikelihood**(*rho2*, *npts*)

Re-normalize correlation.

Convert a squared normalized correlation to a proper log-likelihood associated with a registration problem. The result is a function of both the input correlation and the number of points in the image overlap.

See: Roche, medical image registration through statistical inference, 2001.

Parameters

rho2: float
Squared correlation measure

npts: int
Number of points involved in computing *rho2*

Returns

ll: float
Log-likelihood re-normalized *rho2*

`nipy.algorithms.registration.similarity_measures.`**`dist2loss`**(*q*, *qI=None*, *qJ=None*)

Convert a joint distribution model q(i,j) into a pointwise loss:

L(i,j) = - log q(i,j)/(q(i)q(j))

where q(i) = sum_j q(i,j) and q(j) = sum_i q(i,j)

See: Roche, medical image registration through statistical inference, 2001.

CHAPTER

FORTYNINE

ALGORITHMS.REGISTRATION.TRANSFORM

49.1 Module: `algorithms.registration.transform`

Inheritance diagram for `nipy.algorithms.registration.transform`:

Generic transform class

This implementation specifies an API. We've done our best to avoid checking instances, so any class implementing this API should be valid in the places (like registration routines) that use transforms. If that isn't true, it's a bug.

49.2 `Transform`

class `nipy.algorithms.registration.transform.`**`Transform`**(*func*)

Bases: `object`

A default transformation class

This class specifies the tiny API. That is, the class should implement:

- obj.param - the transformation exposed as a set of parameters. Changing param should change the transformation
- obj.apply(pts) - accepts (N,3) array-like of points in 3 dimensions, returns an (N, 3) array of transformed points
- obj.compose(xform) - accepts another object implementing `apply`, and returns a new transformation object, where the resulting transformation is the composition of the `obj` transform onto the `xform` transform.

`__init__`(*func*)

`apply`(*pts*)

`compose`(*other*)

property param

CHAPTER

FIFTY

ALGORITHMS.REGISTRATION.TYPE_CHECK

50.1 Module: `algorithms.registration.type_check`

Utilities to test whether a variable is of, or convertible to, a particular type

50.2 Functions

`nipy.algorithms.registration.type_check.`**`check_type`**`(`*x*, *t*, *accept_none=False*`)`

Checks whether a variable is convertible to a certain type. A ValueError is raised if test fails.

Parameters

x
[object] Input argument to be checked.

t
[type] Target type.

accept_none
[bool] If True, skip errors if *x* is None.

`nipy.algorithms.registration.type_check.`**`check_type_and_shape`**`(`*x*, *t*, *s*, *accept_none=False*`)`

Checks whether a sequence is convertible to a numpy ndarray with given shape, and if the elements are convertible to a certain type. A ValueError is raised if test fails.

Parameters

x
[sequence] Input sequence to be checked.

t
[type] Target element-wise type.

s
[sequence of ints] Target shape.

accept_none
[bool] If True, skip errors if *x* is None.

CHAPTER

FIFTYONE

ALGORITHMS.RESAMPLE

51.1 Module: `algorithms.resample`

Some simple examples and utility functions for resampling.

51.2 Functions

`nipy.algorithms.resample.`**`resample`**(*image*, *target*, *mapping*, *shape*, *order=3*, *mode='constant'*, *cval=0.0*)

Resample *image* to *target* CoordinateMap

Use a "world-to-world" mapping *mapping* and spline interpolation of a *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] image that is to be resampled.

target
[CoordinateMap] coordinate map for output image.

mapping
[callable or tuple or array] transformation from target.function_range to image.coordmap.function_range, 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 mapping as an affine array, in homogeneous coordinates.

shape
[sequence of int] shape of output array, in target.function_domain.

order
[int, optional] what order of interpolation to use in `scipy.ndimage`.

mode
[str, optional] Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect' or 'wrap'). Default is 'constant'.

cval
[scalar, optional] Value used for points outside the boundaries of the input if mode='constant'. Default is 0.0.

Returns

output
[Image instance] Image has interpolated data and output.coordmap == target.

`nipy.algorithms.resample.`**`resample_img2img`**(*source*, *target*, *order=3*, *mode='constant'*, *cval=0.0*)

Resample *source* image to space of *target* image

This wraps the resample function to resample one image onto another. The output of the function will give an image with shape of the target and data from the source.

Parameters

source
[`Image`] Image instance that is to be resampled

target
[`Image`] Image instance to which source is resampled. The output image will have the same shape as the target, and the same coordmap.

order
[`int`, optional] What order of interpolation to use in `scipy.ndimage`.

mode
[str, optional] Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect' or 'wrap'). Default is 'constant'.

cval
[scalar, optional] Value used for points outside the boundaries of the input if mode='constant'. Default is 0.0.

Returns

output
[`Image`] Image with interpolated data and output.coordmap == target.coordmap

Examples

```
>>> from nipy.testing import funcfile, anatfile
>>> from nipy.io.api import load_image
>>> aimg_source = load_image(anatfile)
>>> aimg_target = aimg_source
>>> # in this case, we resample aimg to itself
>>> resimg = resample_img2img(aimg_source, aimg_target)
```

CHAPTER FIFTYTWO

ALGORITHMS.SEGMENTATION.BRAIN_SEGMENTATION

52.1 Module: `algorithms.segmentation.brain_segmentation`

Inheritance diagram for `nipy.algorithms.segmentation.brain_segmentation`:

52.2 `BrainT1Segmentation`

class `nipy.algorithms.segmentation.brain_segmentation.`**`BrainT1Segmentation`**(*data*, *mask=None*, *model='3k'*, *niters=25*, *ngb_size=6*, *beta=0.5*, *ref_params=None*, *init_params=None*, *convert=True*)

Bases: `object`

`__init__`(*data*, *mask=None*, *model='3k'*, *niters=25*, *ngb_size=6*, *beta=0.5*, *ref_params=None*, *init_params=None*, *convert=True*)

`convert`()

CHAPTER

FIFTYTHREE

ALGORITHMS.SEGMENTATION.SEGMENTATION

53.1 Module: `algorithms.segmentation.segmentation`

Inheritance diagram for `nipy.algorithms.segmentation.segmentation`:

53.2 Class

53.3 `Segmentation`

class `nipy.algorithms.segmentation.segmentation.`**`Segmentation`**(*data*, *mask=None*, *mu=None*, *sigma=None*, *ppm=None*, *prior=None*, *U=None*, *ngb_size=26*, *beta=0.1*)

Bases: `object`

`__init__`(*data*, *mask=None*, *mu=None*, *sigma=None*, *ppm=None*, *prior=None*, *U=None*, *ngb_size=26*, *beta=0.1*)

Class for multichannel Markov random field image segmentation using the variational EM algorithm. For details regarding the underlying algorithm, see:

Roche et al, 2011. On the convergence of EM-like algorithms for image segmentation using Markov random fields. Medical Image Analysis (DOI: 10.1016/j.media.2011.05.002).

Parameters

data
[array-like] Input image array

mask
[array-like or tuple of array] Input mask to restrict the segmentation

beta
[float] Markov regularization parameter

mu
[array-like] Initial class-specific means

sigma
[array-like] Initial class-specific variances

`free_energy`(*ppm=None*)

Compute the free energy defined as:

F(q, theta) = int q(x) log q(x)/p(x,y/theta) dx

associated with input parameters mu, sigma and beta (up to an ignored constant).

`log_external_field`()

Compute the logarithm of the external field, where the external field is defined as the likelihood times the first-order component of the prior.

`map`()

Return the maximum a posterior label map

`normalized_external_field`()

`run`(*niters=10*, *freeze=()*)

`set_markov_prior`(*beta*, *U=None*)

`ve_step`()

`vm_step`(*freeze=()*)

53.4 Functions

`nipy.algorithms.segmentation.segmentation.binarize_ppm`(*q*)

Assume input ppm is masked (ndim==2)

`nipy.algorithms.segmentation.segmentation.map_from_ppm`(*ppm*, *mask=None*)

`nipy.algorithms.segmentation.segmentation.moment_matching`(*dat*, *mu*, *sigma*, *glob_mu*, *glob_sigma*)

Moment matching strategy for parameter initialization to feed a segmentation algorithm.

Parameters

data: array
Image data.

mu
[array] Template class-specific intensity means

sigma
[array] Template class-specific intensity variances

glob_mu
[float] Template global intensity mean

glob_sigma
[float] Template global intensity variance

Returns

dat_mu: array
Guess of class-specific intensity means

dat_sigma: array
Guess of class-specific intensity variances

CHAPTER

FIFTYFOUR

ALGORITHMS.SLICETIMING.TIMEFUNCS

54.1 Module: `algorithms.slicetiming.timefuncs`

Utility functions for returning slice times from number of slices and TR

Slice timing routines in nipy need a vector of slice times.

Slice times are vectors t_i with $i = 0...N$ of times, one for each slice, where t_i gives the time at which slice number i was acquired, relative to the beginning of the volume acquisition.

We like these vectors because they are unambiguous; the indices i refer to positions in space, and the values t_i refer to times.

But, there are many common slice timing regimes for which it's easy to get the slice times once you know the volume acquisition time (the TR) and the number of slices.

For example, if you acquired the slices in a simple ascending order, and you have 10 slices and the TR was 2.0, then the slice times are:

```
>>> import numpy as np
>>> np.arange(10) / 10.  * 2.0
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8])
```

These are small convenience functions that accept the number of slices and the TR as input, and return a vector of slice times:

```
>>> ascending(10, 2.)
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8])
```

54.2 Functions

`nipy.algorithms.slicetiming.timefuncs.`**`st_01234`**(*n_slices*, *TR*)

Simple ascending slice sequence

slice 0 first, slice 1 second etc.

For example, for 5 slices and a TR of 1:

```
>>> st_01234(5, 1.)
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Note: slice 0 is the first slice in the voxel data block

Parameters

n_slices
[int] Number of slices in volume

TR
[float] Time to acquire one full volume

Returns

slice_times
[(n_slices,) ndarray] Vectors $t_i i = 0 \ldots N$ of times, one for each slice, where t_i gives the time at which slice number i was acquired, relative to the beginning of the volume acquisition.

`nipy.algorithms.slicetiming.timefuncs.`**`st_02413`**`(`*n_slices*, *TR*`)`

Ascend alternate every second slice, starting at first slice

Collect slice 0 first, slice 2 second up to top. Then return to collect slice 1, slice 3 etc.

For example, for 5 slices and a TR of 1:

```
>>> st_02413(5, 1.)
array([ 0. ,  0.6,  0.2,  0.8,  0.4])
```

Note: slice 0 is the first slice in the voxel data block

Parameters

n_slices
[int] Number of slices in volume

TR
[float] Time to acquire one full volume

Returns

slice_times
[(n_slices,) ndarray] Vectors $t_i i = 0 \ldots N$ of times, one for each slice, where t_i gives the time at which slice number i was acquired, relative to the beginning of the volume acquisition.

`nipy.algorithms.slicetiming.timefuncs.`**`st_03142`**`(`*n_slices*, *TR*`)`

Ascend alternate, where alternation is by half the volume

Collect slice 0 then slice `ceil(n_slices / 2.)` then slice 1 then slice `ceil(nslices / 2.) + 1` etc.

For example, for 5 slices and a TR of 1:

```
>>> st_03142(5, 1.)
array([ 0. ,  0.4,  0.8,  0.2,  0.6])
```

Note: slice 0 is the first slice in the voxel data block

Parameters

n_slices
[int] Number of slices in volume

TR
[float] Time to acquire one full volume

Returns

slice_times
[(n_slices,) ndarray] Vectors $t_i i = 0 ... N$ of times, one for each slice, where t_i gives the time at which slice number i was acquired, relative to the beginning of the volume acquisition.

`nipy.algorithms.slicetiming.timefuncs.`**`st_13024`**`(`*n_slices*, *TR*`)`

Ascend alternate every second slice, starting at second slice

Collect slice 1 first, slice 3 second up to top (highest numbered slice). Then return to collect slice 0, slice 2 etc. This order is rare except on Siemens acquisitions with an even number of slices. See *`st_odd0_even1()`* for this logic.

For example, for 5 slices and a TR of 1:

```
>>> st_13024(5, 1.)
array([ 0.4,  0. ,  0.6,  0.2,  0.8])
```

Note: slice 0 is the first slice in the voxel data block

Parameters

n_slices
[int] Number of slices in volume

TR
[float] Time to acquire one full volume

Returns

slice_times
[(n_slices,) ndarray] Vectors $t_i i = 0 ... N$ of times, one for each slice, where t_i gives the time at which slice number i was acquired, relative to the beginning of the volume acquisition.

`nipy.algorithms.slicetiming.timefuncs.`**`st_41302`**`(`*n_slices*, *TR*`)`

Descend alternate, where alternation is by half the volume

Collect slice `(n_slices - 1)` then slice `floor(nslices / 2.) - 1` then slice `(n_slices - 2)` then slice `floor(nslices / 2.) - 2` etc.

For example, for 5 slices and a TR of 1:

```
>>> st_41302(5, 1.)
array([ 0.6,  0.2,  0.8,  0.4,  0. ])
```

Note: slice 0 is the first slice in the voxel data block

Parameters

n_slices
[int] Number of slices in volume

TR
[float] Time to acquire one full volume

Returns

slice_times
[(n_slices,) ndarray] Vectors $t_i i = 0 ... N$ of times, one for each slice, where t_i gives the time at which slice number i was acquired, relative to the beginning of the volume acquisition.

`nipy.algorithms.slicetiming.timefuncs.`**`st_42031`**`(`*n_slices*, *TR*`)`

Descend alternate every second slice, starting at last slice

Collect slice (*n_slices* - 1) first, slice (*nslices* - 3) second down to bottom (lowest numbered slice). Then return to collect slice (*n_slices* -2), slice (*n_slices* - 4) etc.

For example, for 5 slices and a TR of 1:

```
>>> st_42031(5, 1.)
array([ 0.4,  0.8,  0.2,  0.6,  0. ])
```

Note: slice 0 is the first slice in the voxel data block

Parameters

n_slices
[int] Number of slices in volume

TR
[float] Time to acquire one full volume

Returns

slice_times
[(n_slices,) ndarray] Vectors $t_i i = 0 \ldots N$ of times, one for each slice, where t_i gives the time at which slice number i was acquired, relative to the beginning of the volume acquisition.

`nipy.algorithms.slicetiming.timefuncs.`**`st_43210`**`(`*n_slices*, *TR*`)`

Simple descending slice sequence

slice `n_slices-1` first, slice `n_slices - 2` second etc.

For example, for 5 slices and a TR of 1:

```
>>> st_43210(5, 1.)
array([ 0.8,  0.6,  0.4,  0.2,  0. ])
```

Note: slice 0 is the first slice in the voxel data block

Parameters

n_slices
[int] Number of slices in volume

TR
[float] Time to acquire one full volume

Returns

slice_times
[(n_slices,) ndarray] Vectors $t_i i = 0 \ldots N$ of times, one for each slice, where t_i gives the time at which slice number i was acquired, relative to the beginning of the volume acquisition.

`nipy.algorithms.slicetiming.timefuncs.`**`st_odd0_even1`**`(`*n_slices*, *TR*`)`

Ascend alternate starting at slice 0 for odd, slice 1 for even *n_slices*

Acquisitions with alternating ascending slices from Siemens scanners often seem to have this behavior as default - see:

https://mri.radiology.uiowa.edu/fmri_images.html

This means we use the `st_02413()` algorithm if *n_slices* is odd, and the `st_13024()` algorithm if *n_slices* is even.

For example, for 4 slices and a TR of 1:

```
>>> st_odd0_even1(4, 1.)
array([ 0.5 ,  0.  ,  0.75,  0.25])
```

5 slices and a TR of 1:

```
>>> st_odd0_even1(5, 1.)
array([ 0. ,  0.6,  0.2,  0.8,  0.4])
```

Note: slice 0 is the first slice in the voxel data block

Parameters

n_slices
[int] Number of slices in volume

TR
[float] Time to acquire one full volume

Returns

slice_times
[(n_slices,) ndarray] Vectors $t_i i = 0...N$ of times, one for each slice, where t_i gives the time at which slice number i was acquired, relative to the beginning of the volume acquisition.

CHAPTER

FIFTYFIVE

ALGORITHMS.STATISTICS.BAYESIAN_MIXED_EFFECTS

55.1 Module: `algorithms.statistics.bayesian_mixed_effects`

Generic implementation of multiple regression analysis under noisy measurements.

`nipy.algorithms.statistics.bayesian_mixed_effects.`**`two_level_glm`**(*y*, *vy*, *X*, *niter=10*)

Inference of a mixed-effect linear model using the variational Bayes algorithm.

Parameters

y
[array-like] Array of observations. Shape should be (n, ...) where n is the number of independent observations per unit.

vy
[array-like] First-level variances associated with the observations. Should be of the same shape as Y.

X
[array-like] Second-level design matrix. Shape should be (n, p) where n is the number of observations per unit, and p is the number of regressors.

Returns

beta
[array-like] Effect estimates (posterior means)

s2
[array-like] Variance estimates. The posterior variance matrix of beta[:, i] may be computed by s2[:, i] * inv(X.T * X)

dof
[float] Degrees of freedom as per the variational Bayes approximation (simply, the number of observations minus the number of independent regressors)

CHAPTER FIFTYSIX

ALGORITHMS.STATISTICS.BENCH.BENCH_INTVOL

56.1 Module: `algorithms.statistics.bench.bench_intvol`

56.2 Functions

`nipy.algorithms.statistics.bench.bench_intvol.`**`bench_lips1d`**`()`

`nipy.algorithms.statistics.bench.bench_intvol.`**`bench_lips2d`**`()`

`nipy.algorithms.statistics.bench.bench_intvol.`**`bench_lips3d`**`()`

CHAPTER

FIFTYSEVEN

ALGORITHMS.STATISTICS.EMPIRICAL_PVALUE

57.1 Module: `algorithms.statistics.empirical_pvalue`

Inheritance diagram for `nipy.algorithms.statistics.empirical_pvalue`:

Routines to get corrected p-values estimates, based on the observations.

It implements 3 approaches:

- Benjamini-Hochberg FDR: http://en.wikipedia.org/wiki/False_discovery_rate
- a class that fits a Gaussian model to the central part of an histogram, following [1]

  [1] Schwartzman A, Dougherty RF, Lee J, Ghahremani D, Taylor JE. Empirical null and false discovery rate analysis in neuroimaging. Neuroimage. 2009 Jan 1;44(1):71-82. PubMed PMID: 18547821. DOI: 10.1016/j.neuroimage.2008.04.182

  This is typically necessary to estimate a FDR when one is not certain that the data behaves as a standard normal under H_0.
- a model based on Gaussian mixture modelling 'a la Oxford'

Author : Bertrand Thirion, Yaroslav Halchenko, 2008-2012

57.2 Class

57.3 `NormalEmpiricalNull`

class `nipy.algorithms.statistics.empirical_pvalue.`**`NormalEmpiricalNull`**(*x*)

Bases: `object`

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*)

Initialize 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

Parameters

theta
[float or array of shape (n_samples)] values to test

Returns

afp
[value of array of shape(n)]

`fdrcurve`()

Returns the FDR associated with any point of self.x

`learn`(*left=0.2*, *right=0.8*)

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: variance of the estimated normal distribution
- sigma: np.sqrt(sqsigma) : standard deviation of the estimated normal distribution

`plot`(*efp=None*, *alpha=0.05*, *bar=1*, *mpaxes=None*)

Plot the histogram of x

Parameters

efp
[float, optional] The empirical FDR (corresponding to x) if efp==None, the false positive rate threshold plot is not drawn.

alpha
[float, optional] The chosen FDR threshold

bar=1
[bool, optional]

mpaxes=None: if not None, handle to an axes where the fig will be drawn. Avoids creating unnecessarily new figures

`threshold`(*alpha=0.05*, *verbose=0*)

Compute the threshold corresponding 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.

Returns

theta: float
the critical value associated with the provided FDR

`uncorrected_threshold`(*alpha=0.001*, *verbose=0*)

Compute the threshold corresponding to a specificity alpha for x

Parameters

alpha
[float, optional] the chosen false discovery rate (FDR) threshold.

verbose
[boolean, optional] the verbosity level, if True a plot is generated.

Returns

theta: float
the critical value associated with the provided p-value

57.4 Functions

`nipy.algorithms.statistics.empirical_pvalue.check_p_values`(*p_values*)

Basic checks on the p_values array: values should be within [0,1]

Assures also that p_values are at least in 1d array. None of the checks is performed if p_values is None.

Parameters

p_values
[array of shape (n)] The sample p-values

Returns

p_values
[array of shape (n)] The sample p-values

`nipy.algorithms.statistics.empirical_pvalue.fdr`(*p_values=None*, *verbose=0*)

Returns the FDR associated with each p value

Parameters

p_values
[ndarray of shape (n)] The samples p-value

Returns

q
[array of shape(n)] The corresponding fdr values

`nipy.algorithms.statistics.empirical_pvalue.`**`fdr_threshold`**(*p_values*, *alpha=0.05*)

Return FDR threshold given p values

Parameters

p_values
[array of shape (n), optional] The samples p-value

alpha
[float, optional] The desired FDR significance

Returns

critical_p_value: float
The p value corresponding to the FDR alpha

`nipy.algorithms.statistics.empirical_pvalue.`**`gamma_gaussian_fit`**(*x*, *test=None*, *verbose=0*, *mpaxes=False*, *bias=1*, *gaussian_mix=0*, *return_estimator=False*)

Computing some prior probabilities that the voxels of a certain map are in class disactivated, null or active using a gamma-Gaussian mixture

Parameters

x: array of shape (nvox,)
the map to be analysed

test: array of shape (nbitems,), optional
the test values for which the p-value needs to be computed by default, test = x

verbose: 0, 1 or 2, optional
verbosity mode, 0 is quiet, and 2 calls matplotlib to display graphs.

mpaxes: matplotlib axes, optional
axes handle used to plot the figure in verbose mode if None, new axes are created if false, nothing is done

bias: float, optional
lower bound on the Gaussian variance (to avoid shrinkage)

gaussian_mix: float, optional
if nonzero, lower bound on the Gaussian mixing weight (to avoid shrinkage)

return_estimator: boolean, optional
if return_estimator is true, the estimator object is returned.

Returns

bfp: array of shape (nbitems,3)
The probability of each component in the mixture model for each test value

estimator: nipy.labs.clustering.ggmixture.GGGM object
The estimator object, returned only if return_estimator is true.

`nipy.algorithms.statistics.empirical_pvalue.`**`gaussian_fdr`**(*x*)

Return the FDR associated with each value assuming a Gaussian distribution `nipy.algorithms.statistics.empirical_pvalue.`**`gaussian_fdr_threshold`**(*x*, *alpha=0.05*)

Return FDR threshold given normal variates

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

x: ndarray
input data

alpha: float, optional
desired significance

Returns

threshold
[float] threshold, given as a Gaussian critical value

`nipy.algorithms.statistics.empirical_pvalue.`**`smoothed_histogram_from_samples`**(*x*, *bins=None*, *nbins=256*, *normalized=False*)

Smooth histogram corresponding to density underlying the samples in *x*

Parameters

x: array of shape(n_samples)
input data

bins: array of shape(nbins+1), optional
the bins location

nbins: int, optional
the number of bins of the resulting histogram

normalized: bool, optional
if True, the result is returned as a density value

Returns

h: array of shape (nbins)
the histogram

bins: array of shape(nbins+1),
the bins location

`nipy.algorithms.statistics.empirical_pvalue.`**`three_classes_GMM_fit`**(*x*, *test=None*, *alpha=0.01*, *prior_strength=100*, *verbose=0*, *fixed_scale=False*, *mpaxes=None*, *bias=0*, *theta=0*, *return_estimator=False*)

Fit the data with a 3-classes Gaussian Mixture Model, i.e. compute some probability that the voxels of a certain map are in class disactivated, null or active

Parameters

x: array of shape (nvox,1)
The map to be analysed

test: array of shape(nbitems,1), optional
the test values for which the p-value needs to be computed by default (if None), test=x

alpha: float, optional
the prior weights of the positive and negative classes

prior_strength: float, optional
the confidence on the prior (should be compared to size(x))

verbose: int
verbosity mode

fixed_scale: bool, optional
boolean, variance parameterization. if True, the variance is locked to 1 otherwise, it is estimated from the data

mpaxes:
axes handle used to plot the figure in verbose mode if None, new axes are created

bias: bool
allows a rescaling of the posterior probability that takes into account the threshold theta. Not rigorous.

theta: float
the threshold used to correct the posterior p-values when bias=1; normally, it is such that test>theta note that if theta = -np.inf, the method has a standard behaviour

return_estimator: boolean, optional
If return_estimator is true, the estimator object is returned.

Returns

bfp
[array of shape (nbitems,3):] the posterior probability of each test item belonging to each component in the GMM (sum to 1 across the 3 classes) if np.size(test)==0, i.e. nbitem==0, None is returned

estimator
[nipy.labs.clustering.GMM object] The estimator object, returned only if return_estimator is true.

Notes

Our convention is that:

- class 1 represents the negative class
- class 2 represents the null class
- class 3 represents the positive class

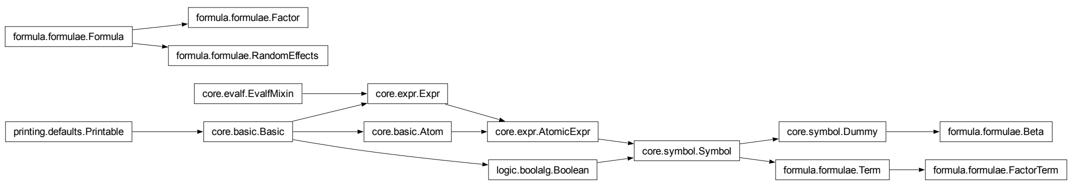
CHAPTER

FIFTYEIGHT

ALGORITHMS.STATISTICS.FORMULA.FORMULAE

58.1 Module: `algorithms.statistics.formula.formulae`

Inheritance diagram for `nipy.algorithms.statistics.formula.formulae`:



58.1.1 Formula objects

A formula is basically a sympy expression for the mean of something of the form:

```
mean = sum([Beta(e)*e for e in expr])
```

Or, a linear combination of sympy expressions, with each one multiplied by its own "Beta". The elements of expr can be instances of Term (for a linear regression formula, they would all be instances of Term). But, in general, there might be some other parameters (i.e. sympy.Symbol instances) that are not Terms.

The design matrix is made up of columns that are the derivatives of mean with respect to everything that is not a Term, evaluated at a recarray that has field names given by [str(t) for t in self.terms].

For those familiar with R's formula syntax, if we wanted a design matrix like the following:

```
> s.table = read.table("http://www-stat.stanford.edu/~jtaylo/courses/stats191/data/
↪supervisor.table", header=T)
> d = model.matrix(lm(Y ~ X1*X3, s.table)
)
> d
   (Intercept) X1 X3 X1:X3
1            1 51 39  1989
2            1 64 54  3456
```

(continues on next page)

(continued from previous page)

```
3             1 70 69  4830
4             1 63 47  2961
5             1 78 66  5148
6             1 55 44  2420
7             1 67 56  3752
8             1 75 55  4125
9             1 82 67  5494
10            1 61 47  2867
11            1 53 58  3074
12            1 60 39  2340
13            1 62 42  2604
14            1 83 45  3735
15            1 77 72  5544
16            1 90 72  6480
17            1 85 69  5865
18            1 60 75  4500
19            1 70 57  3990
20            1 58 54  3132
21            1 40 34  1360
22            1 61 62  3782
23            1 66 50  3300
24            1 37 58  2146
25            1 54 48  2592
26            1 77 63  4851
27            1 75 74  5550
28            1 57 45  2565
29            1 85 71  6035
30            1 82 59  4838
attr(,"assign")
[1] 0 1 2 3
>
```

With the Formula, it looks like this:

```
>>> r = np.rec.array([
...     (43, 51, 30, 39, 61, 92, 45), (63, 64, 51, 54, 63, 73, 47),
...     (71, 70, 68, 69, 76, 86, 48), (61, 63, 45, 47, 54, 84, 35),
...     (81, 78, 56, 66, 71, 83, 47), (43, 55, 49, 44, 54, 49, 34),
...     (58, 67, 42, 56, 66, 68, 35), (71, 75, 50, 55, 70, 66, 41),
...     (72, 82, 72, 67, 71, 83, 31), (67, 61, 45, 47, 62, 80, 41),
...     (64, 53, 53, 58, 58, 67, 34), (67, 60, 47, 39, 59, 74, 41),
...     (69, 62, 57, 42, 55, 63, 25), (68, 83, 83, 45, 59, 77, 35),
...     (77, 77, 54, 72, 79, 77, 46), (81, 90, 50, 72, 60, 54, 36),
...     (74, 85, 64, 69, 79, 79, 63), (65, 60, 65, 75, 55, 80, 60),
...     (65, 70, 46, 57, 75, 85, 46), (50, 58, 68, 54, 64, 78, 52),
...     (50, 40, 33, 34, 43, 64, 33), (64, 61, 52, 62, 66, 80, 41),
...     (53, 66, 52, 50, 63, 80, 37), (40, 37, 42, 58, 50, 57, 49),
...     (63, 54, 42, 48, 66, 75, 33), (66, 77, 66, 63, 88, 76, 72),
...     (78, 75, 58, 74, 80, 78, 49), (48, 57, 44, 45, 51, 83, 38),
...     (85, 85, 71, 71, 77, 74, 55), (82, 82, 39, 59, 64, 78, 39)],
...              dtype=[('y', '<i8'), ('x1', '<i8'), ('x2', '<i8'),
...                     ('x3', '<i8'), ('x4', '<i8'), ('x5', '<i8'),
```

(continues on next page)

(continued from previous page)

```
...                      ('x6', '<i8')])
>>> x1 = Term('x1'); x3 = Term('x3')
>>> f = Formula([x1, x3, x1*x3]) + I
>>> f.mean
_b0*x1 + _b1*x3 + _b2*x1*x3 + _b3
```

The I is the "intercept" term, I have explicitly not used R's default of adding it to everything.

```
>>> f.design(r)
array([(51.0, 39.0, 1989.0, 1.0), (64.0, 54.0, 3456.0, 1.0),
       (70.0, 69.0, 4830.0, 1.0), (63.0, 47.0, 2961.0, 1.0),
       (78.0, 66.0, 5148.0, 1.0), (55.0, 44.0, 2420.0, 1.0),
       (67.0, 56.0, 3752.0, 1.0), (75.0, 55.0, 4125.0, 1.0),
       (82.0, 67.0, 5494.0, 1.0), (61.0, 47.0, 2867.0, 1.0),
       (53.0, 58.0, 3074.0, 1.0), (60.0, 39.0, 2340.0, 1.0),
       (62.0, 42.0, 2604.0, 1.0), (83.0, 45.0, 3735.0, 1.0),
       (77.0, 72.0, 5544.0, 1.0), (90.0, 72.0, 6480.0, 1.0),
       (85.0, 69.0, 5865.0, 1.0), (60.0, 75.0, 4500.0, 1.0),
       (70.0, 57.0, 3990.0, 1.0), (58.0, 54.0, 3132.0, 1.0),
       (40.0, 34.0, 1360.0, 1.0), (61.0, 62.0, 3782.0, 1.0),
       (66.0, 50.0, 3300.0, 1.0), (37.0, 58.0, 2146.0, 1.0),
       (54.0, 48.0, 2592.0, 1.0), (77.0, 63.0, 4851.0, 1.0),
       (75.0, 74.0, 5550.0, 1.0), (57.0, 45.0, 2565.0, 1.0),
       (85.0, 71.0, 6035.0, 1.0), (82.0, 59.0, 4838.0, 1.0)],
      dtype=[('x1', '<f8'), ('x3', '<f8'), ('x1*x3', '<f8'), ('1', '<f8')])
```

58.2 Classes

58.2.1 Beta

class nipy.algorithms.statistics.formula.formulae.**Beta**(*name*, *term*)

Bases: Dummy

A symbol tied to a Term *term*

__init__(**args*, ***kwargs*)

adjoint()

apart(*x=None*, ***args*)

See the apart function in sympy.polys

property args: tuple[Basic, ...]

Returns a tuple of arguments of 'self'.

Notes

Never use self._args, always use self.args. Only use _args in __new__ when creating a new function. Do not override .args() from Basic (so that it is easy to change the interface in the future if needed).

Examples

```
>>> from sympy import cot
>>> from sympy.abc import x, y
```

```
>>> cot(x).args
(x,)
```

```
>>> cot(x).args[0]
x
```

```
>>> (x*y).args
(x, y)
```

```
>>> (x*y).args[1]
y
```

args_cnc(*cset=False*, *warn=True*, *split_1=True*)

Return [commutative factors, non-commutative factors] of self.

Examples

```
>>> from sympy import symbols, oo
>>> A, B = symbols('A B', commutative=0)
>>> x, y = symbols('x y')
>>> (-2*x*y).args_cnc()
[[-1, 2, x, y], []]
>>> (-2.5*x).args_cnc()
[[-1, 2.5, x], []]
>>> (-2*x*A*B*y).args_cnc()
[[-1, 2, x, y], [A, B]]
>>> (-2*x*A*B*y).args_cnc(split_1=False)
[[-2, x, y], [A, B]]
>>> (-2*x*y).args_cnc(cset=True)
[{-1, 2, x, y}, []]
```

The arg is always treated as a Mul:

```
>>> (-2 + x + A).args_cnc()
[[], [x - 2 + A]]
>>> (-oo).args_cnc() # -oo is a singleton
[[-1, oo], []]
```

as_base_exp() → tuple[Expr, Expr]

as_coeff_Add(*rational=False*) → tuple[Number, Expr]

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational: bool = False*) → tuple[Number, Expr]

Efficiently extract the coefficient of a product.

as_coeff_add(**deps*) → tuple[Expr, tuple[Expr, ...]]

Return the tuple (c, args) where self is written as an Add, `a`.

c should be a Rational added to any terms of the Add that are independent of deps.

args should be a tuple of all other terms of `a`; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you do not know if self is an Add or not but you want to treat self as an Add or if you want to process the individual arguments of the tail of self as an Add.

- if you know self is an Add and want only the head, use self.args[0];
- if you do not want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail.
- if you want to split self into an independent and dependent parts use `self.as_independent(*deps)`

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_add()
(3, ())
>>> (3 + x).as_coeff_add()
(3, (x,))
>>> (3 + x + y).as_coeff_add(x)
(y + 3, (x,))
>>> (3 + y).as_coeff_add(x)
(y + 3, ())
```

as_coeff_exponent(*x*) → tuple[Expr, Expr]

`c*x**e -> c,e` where x can be any symbolic expression.

as_coeff_mul(**deps*, ***kwargs*) → tuple[Expr, tuple[Expr, ...]]

Return the tuple (c, args) where self is written as a Mul, `m`.

c should be a Rational multiplied by any factors of the Mul that are independent of deps.

args should be a tuple of all other factors of m; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you do not know if self is a Mul or not but you want to treat self as a Mul or if you want to process the individual arguments of the tail of self as a Mul.

- if you know self is a Mul and want only the head, use self.args[0];
- if you do not want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail;
- if you want to split self into an independent and dependent parts use `self.as_independent(*deps)`

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_mul()
(3, ())
```

(continues on next page)

(continued from previous page)

```
>>> (3*x*y).as_coeff_mul()
(3, (x, y))
>>> (3*x*y).as_coeff_mul(x)
(3*y, (x,))
>>> (3*y).as_coeff_mul(x)
(3*y, ())
```

as_coefficient(*expr*)

Extracts symbolic coefficient at the given expression. In other words, this functions separates 'self' into the product of 'expr' and 'expr'-free coefficient. If such separation is not possible it will return None.

See also:

coeff
return sum of terms have a given factor

as_coeff_Add
separate the additive constant from an expression

as_coeff_Mul
separate the multiplicative constant from an expression

as_independent
separate x-dependent terms/factors from others

sympy.polys.polytools.Poly.coeff_monomial
efficiently find the single coefficient of a monomial in Poly

sympy.polys.polytools.Poly.nth
like coeff_monomial but powers of monomial terms are used

Examples

```
>>> from sympy import E, pi, sin, I, Poly
>>> from sympy.abc import x
```

```
>>> E.as_coefficient(E)
1
>>> (2*E).as_coefficient(E)
2
>>> (2*sin(E)*E).as_coefficient(E)
```

Two terms have E in them so a sum is returned. (If one were desiring the coefficient of the term exactly matching E then the constant from the returned expression could be selected. Or, for greater precision, a method of Poly can be used to indicate the desired term from which the coefficient is desired.)

```
>>> (2*E + x*E).as_coefficient(E)
x + 2
>>> _.args[0]  # just want the exact match
2
>>> p = Poly(2*E + x*E); p
Poly(x*E + 2*E, x, E, domain='ZZ')
>>> p.coeff_monomial(E)
2
```

(continues on next page)

(continued from previous page)

```
>>> p.nth(0, 1)
2
```

Since the following cannot be written as a product containing E as a factor, None is returned. (If the coefficient `2*x` is desired then the `coeff` method should be used.)

```
>>> (2*E*x + x).as_coefficient(E)
>>> (2*E*x + x).coeff(E)
2*x
```

```
>>> (E*(x + 1) + x).as_coefficient(E)
```

```
>>> (2*pi*I).as_coefficient(pi*I)
2
>>> (2*I).as_coefficient(pi*I)
```

as_coefficients_dict(**syms*)

Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0.

If symbols `syms` are provided, any multiplicative terms independent of them will be considered a coefficient and a regular dictionary of syms-dependent generators as keys and their corresponding coefficients as values will be returned.

Examples

```
>>> from sympy.abc import a, x, y
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
>>> (3*a*x).as_coefficients_dict(x)
{x: 3*a}
>>> (3*a*x).as_coefficients_dict(y)
{1: 3*a*x}
```

as_content_primitive(*radical=False*, *clear=True*)

This method should recursively remove a Rational from all arguments and return that (content) and the new self (primitive). The content should always be positive and `Mul(*foo.as_content_primitive()) == foo`. The primitive need not be in canonical form and should try to preserve the underlying structure if possible (i.e. expand_mul should not be applied to self).

Examples

```
>>> from sympy import sqrt
>>> from sympy.abc import x, y, z
```

```
>>> eq = 2 + 2*x + 2*y*(3 + 3*y)
```

The as_content_primitive function is recursive and retains structure:

```
>>> eq.as_content_primitive()
(2, x + 3*y*(y + 1) + 1)
```

Integer powers will have Rationals extracted from the base:

```
>>> ((2 + 6*x)**2).as_content_primitive()
(4, (3*x + 1)**2)
>>> ((2 + 6*x)**(2*y)).as_content_primitive()
(1, (2*(3*x + 1))**(2*y))
```

Terms may end up joining once their as_content_primitives are added:

```
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(11, x*(y + 1))
>>> ((3*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(9, x*(y + 1))
>>> ((3*(z*(1 + y)) + 2.0*x*(3 + 3*y))).as_content_primitive()
(1, 6.0*x*(y + 1) + 3*z*(y + 1))
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))**2).as_content_primitive()
(121, x**2*(y + 1)**2)
>>> ((x*(1 + y) + 0.4*x*(3 + 3*y))**2).as_content_primitive()
(1, 4.84*x**2*(y + 1)**2)
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

If clear=False (default is True) then content will not be removed from an Add if it can be distributed to leave one or more terms with integer coefficients.

```
>>> (x/2 + y).as_content_primitive()
(1/2, x + 2*y)
>>> (x/2 + y).as_content_primitive(clear=False)
(1, x/2 + y)
```

as_dummy()

Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True. When applied to a symbol a new symbol having only the same commutativity will be returned.

Notes

Any object that has structurally bound variables should have a property, *bound_symbols* that returns those symbols appearing in the object.

Examples

```
>>> from sympy import Integral, Symbol
>>> from sympy.abc import x
>>> r = Symbol('r', real=True)
>>> Integral(r, (r, x)).as_dummy()
Integral(_0, (_0, x))
>>> _.variables[0].is_real is None
True
>>> r.as_dummy()
_r
```

as_expr(**gens*)

Convert a polynomial to a SymPy expression.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y
```

```
>>> f = (x**2 + x*y).as_poly(x, y)
>>> f.as_expr()
x**2 + x*y
```

```
>>> sin(x).as_expr()
sin(x)
```

as_independent(**deps*, ***hint*) → tuple[Expr, Expr]

A mostly naive separation of a Mul or Add into arguments that are not are dependent on deps. To obtain as complete a separation of variables as possible, use a separation method first, e.g.:

- separatevars() to change Mul, Add and Pow (including exp) into Mul
- .expand(mul=True) to change Add or Mul into Add
- .expand(log=True) to change log expr into an Add

The only non-naive thing that is done here is to respect noncommutative ordering of variables and to always return (0, 0) for *self* of zero regardless of hints.

For nonzero *self*, the returned tuple (i, d) has the following interpretation:

- i will has no variable that appears in deps
- d will either have terms that contain variables that are in deps, or be equal to 0 (when self is an Add) or 1 (when self is a Mul)
- if self is an Add then self = i + d
- if self is a Mul then self = i*d

- otherwise (self, S.One) or (S.One, self) is returned.

To force the expression to be treated as an Add, use the hint as_Add=True

See also:

`separatevars`
`expand_log`
`sympy.core.add.Add.as_two_terms`
`sympy.core.mul.Mul.as_two_terms`
`as_coeff_mul`

Examples

– self is an Add

```
>>> from sympy import sin, cos, exp
>>> from sympy.abc import x, y, z
```

```
>>> (x + x*y).as_independent(x)
(0, x*y + x)
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> (2*x*sin(x) + y + x + z).as_independent(x)
(y + z, 2*x*sin(x) + x)
>>> (2*x*sin(x) + y + x + z).as_independent(x, y)
(z, 2*x*sin(x) + x + y)
```

– self is a Mul

```
>>> (x*sin(x)*cos(y)).as_independent(x)
(cos(y), x*sin(x))
```

non-commutative terms cannot always be separated out when self is a Mul

```
>>> from sympy import symbols
>>> n1, n2, n3 = symbols('n1 n2 n3', commutative=False)
>>> (n1 + n1*n2).as_independent(n2)
(n1, n1*n2)
>>> (n2*n1 + n1*n2).as_independent(n2)
(0, n1*n2 + n2*n1)
>>> (n1*n2*n3).as_independent(n1)
(1, n1*n2*n3)
>>> (n1*n2*n3).as_independent(n2)
(n1, n2*n3)
>>> ((x-n1)*(x-y)).as_independent(x)
(1, (x - y)*(x - n1))
```

– self is anything else:

```
>>> (sin(x)).as_independent(x)
(1, sin(x))
>>> (sin(x)).as_independent(y)
(sin(x), 1)
```

(continues on next page)

(continued from previous page)

```
>>> exp(x+y).as_independent(x)
(1, exp(x + y))
```

– force self to be treated as an Add:

```
>>> (3*x).as_independent(x, as_Add=True)
(0, 3*x)
```

– force self to be treated as a Mul:

```
>>> (3+x).as_independent(x, as_Add=False)
(1, x + 3)
>>> (-3+x).as_independent(x, as_Add=False)
(1, x - 3)
```

Note how the below differs from the above in making the constant on the dep term positive.

```
>>> (y*(-3+x)).as_independent(x)
(y, x - 3)
```

– use .as_independent() for true independence testing instead
of .has(). The former considers only symbols in the free symbols while the latter considers all symbols

```
>>> from sympy import Integral
>>> I = Integral(x, (x, 1, 2))
>>> I.has(x)
True
>>> x in I.free_symbols
False
>>> I.as_independent(x) == (I, 1)
True
>>> (I + x).as_independent(x) == (I, x)
True
```

Note: when trying to get independent terms, a separation method might need to be used first. In this case, it is important to keep track of what you send to this routine so you know how to interpret the returned values

```
>>> from sympy import separatevars, log
>>> separatevars(exp(x+y)).as_independent(x)
(exp(y), exp(x))
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> separatevars(x + x*y).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).expand(mul=True).as_independent(y)
(x, x*y)
>>> a, b=symbols('a b', positive=True)
>>> (log(a*b).expand(log=True)).as_independent(b)
(log(a), log(b))
```

as_leading_term(**symbols*, *logx=None*, *cdir=0*)

Returns the leading (nonzero) term of the series expansion of self.

The _eval_as_leading_term routines are used to do this, and they must always return a non-zero value.

Examples

```
>>> from sympy.abc import x
>>> (1 + x + x**2).as_leading_term(x)
1
>>> (1/x**2 + x + x**2).as_leading_term(x)
x**(-2)
```

as_numer_denom()

Return the numerator and the denominator of an expression.

expression -> a/b -> a, b

This is just a stub that should be defined by an object's class methods to get anything else.

See also:

normal
: return `a/b` instead of `(a, b)`

as_ordered_factors(*order=None*)

Return list of ordered factors (if Mul) else [self].

as_ordered_terms(*order=None*, *data=False*)

Transform an expression to an ordered list of terms.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.abc import x
```

```
>>> (sin(x)**2*cos(x) + sin(x)**2 + 1).as_ordered_terms()
[sin(x)**2*cos(x), sin(x)**2, 1]
```

as_poly(**gens*, ***args*)

Converts `self` to a polynomial or returns `None`.

as_powers_dict()

Return self as a dictionary of factors with each factor being treated as a power. The keys are the bases of the factors and the values, the corresponding exponents. The resulting dictionary should be used with caution if the expression is a Mul and contains non- commutative factors since the order that they appeared will be lost in the dictionary.

See also:

as_ordered_factors
: An alternative for noncommutative applications, returning an ordered list of factors.

args_cnc
Similar to as_ordered_factors, but guarantees separation of commutative and noncommutative factors.

as_real_imag(*deep=True*, ***hints*)
Performs complex expansion on 'self' and returns a tuple containing collected both real and imaginary parts. This method cannot be confused with re() and im() functions, which does not perform complex expansion at evaluation.

However it is possible to expand both re() and im() functions and get exactly the same results as with a single call to this function.

```
>>> from sympy import symbols, I
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> (x + y*I).as_real_imag()
(x, y)
```

```
>>> from sympy.abc import z, w
```

```
>>> (z + w*I).as_real_imag()
(re(z) - im(w), re(w) + im(z))
```

as_set()
Rewrites Boolean expression in terms of real sets.

Examples

```
>>> from sympy import Symbol, Eq, Or, And
>>> x = Symbol('x', real=True)
>>> Eq(x, 0).as_set()
{0}
>>> (x > 0).as_set()
Interval.open(0, oo)
>>> And(-2 < x, x < 2).as_set()
Interval.open(-2, 2)
>>> Or(x < -2, 2 < x).as_set()
Union(Interval.open(-oo, -2), Interval.open(2, oo))
```

as_terms()
Transform an expression to a list of terms.

aseries(*x=None*, *n=6*, *bound=0*, *hir=False*)
Asymptotic Series expansion of self. This is equivalent to `self.series(x, oo, n)`.

Parameters

self
[Expression] The expression whose series is to be expanded.

x
[Symbol] It is the variable of the expression to be calculated.

n
[Value] The value used to represent the order in terms of `x**n`, up to which the series is to be expanded.

hir
[Boolean] Set this parameter to be True to produce hierarchical series. It stops the recursion at an early level and may provide nicer and more useful results.

bound
[Value, Integer] Use the `bound` parameter to give limit on rewriting coefficients in its normalised form.

Returns

Expr
Asymptotic series expansion of the expression.

See also:

`Expr.aseries`
See the docstring of this function for complete details of this wrapper.

Notes

This algorithm is directly induced from the limit computational algorithm provided by Gruntz. It majorly uses the mrv and rewrite sub-routines. The overall idea of this algorithm is first to look for the most rapidly varying subexpression w of a given expression f and then expands f in a series in w. Then same thing is recursively done on the leading coefficient till we get constant coefficients.

If the most rapidly varying subexpression of a given expression f is f itself, the algorithm tries to find a normalised representation of the mrv set and rewrites f using this normalised representation.

If the expansion contains an order term, it will be either `O(x ** (-n))` or `O(w ** (-n))` where `w` belongs to the most rapidly varying expression of `self`.

References

[1], [2], [3]

Examples

```
>>> from sympy import sin, exp
>>> from sympy.abc import x
```

```
>>> e = sin(1/x + exp(-x)) - sin(1/x)
```

```
>>> e.aseries(x)
(1/(24*x**4) - 1/(2*x**2) + 1 + O(x**(-6), (x, oo)))*exp(-x)
```

```
>>> e.aseries(x, n=3, hir=True)
-exp(-2*x)*sin(1/x)/2 + exp(-x)*cos(1/x) + O(exp(-3*x), (x, oo))
```

```
>>> e = exp(exp(x)/(1 - 1/x))
>>> e.aseries(x)
exp(exp(x)/(1 - 1/x))
```

```
>>> e.aseries(x, bound=3)
exp(exp(x)/x**2)*exp(exp(x)/x)*exp(-exp(x) + exp(x)/(1 - 1/x) - exp(x)/x -␣
↪exp(x)/x**2)*exp(exp(x))
```

For rational expressions this method may return original expression without the Order term. >>> (1/x).aseries(x, n=8) 1/x

property assumptions0

Return object *type* assumptions.

For example:

Symbol('x', real=True) Symbol('x', integer=True)

are different objects. In other words, besides Python type (Symbol in this case), the initial assumptions are also forming their typeinfo.

Examples

```
>>> from sympy import Symbol
>>> from sympy.abc import x
>>> x.assumptions0
{'commutative': True}
>>> x = Symbol("x", positive=True)
>>> x.assumptions0
{'commutative': True, 'complex': True, 'extended_negative': False,
 'extended_nonnegative': True, 'extended_nonpositive': False,
 'extended_nonzero': True, 'extended_positive': True, 'extended_real':
 True, 'finite': True, 'hermitian': True, 'imaginary': False,
 'infinite': False, 'negative': False, 'nonnegative': True,
 'nonpositive': False, 'nonzero': True, 'positive': True, 'real':
 True, 'zero': False}
```

atoms(**types*)

Returns the atoms that form the current object.

By default, only objects that are truly atomic and cannot be divided into smaller pieces are returned: symbols, numbers, and number symbols like I and pi. It is possible to request atoms of any type, however, as demonstrated below.

Examples

```
>>> from sympy import I, pi, sin
>>> from sympy.abc import x, y
>>> (1 + x + 2*sin(y + I*pi)).atoms()
{1, 2, I, pi, x, y}
```

If one or more types are given, the results will contain only those types of atoms.

```
>>> from sympy import Number, NumberSymbol, Symbol
>>> (1 + x + 2*sin(y + I*pi)).atoms(Symbol)
{x, y}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number)
{1, 2}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol)
{1, 2, pi}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol, I)
{1, 2, I, pi}
```

Note that I (imaginary unit) and zoo (complex infinity) are special types of number symbols and are not part of the NumberSymbol class.

The type can be given implicitly, too:

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(x) # x is a Symbol
{x, y}
```

Be careful to check your assumptions when using the implicit option since `S(1).is_Integer = True` but `type(S(1))` is `One`, a special type of SymPy atom, while `type(S(2))` is type `Integer` and will find all integers in an expression:

```
>>> from sympy import S
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(1))
{1}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(2))
{1, 2}
```

Finally, arguments to atoms() can select more than atomic atoms: any SymPy type (loaded in core/__init__.py) can be listed as an argument and those types of "atoms" as found in scanning the arguments of the expression recursively:

```
>>> from sympy import Function, Mul
>>> from sympy.core.function import AppliedUndef
>>> f = Function('f')
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(Function)
{f(x), sin(y + I*pi)}
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(AppliedUndef)
{f(x)}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
{I*pi, 2*sin(y + I*pi)}
```

property binary_symbols

Return from the atoms of self those which are free symbols.

Not all free symbols are `Symbol`. Eg: IndexedBase('I')[0].free_symbols

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

cancel(**gens*, ***args*)

See the cancel function in sympy.polys

property canonical_variables

Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

Examples

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> Lambda(x, 2*x).canonical_variables
{x: _0}
```

classmethod class_key()

Nice order of classes.

coeff(*x*, *n=1*, *right=False*, *_first=True*)

Returns the coefficient from the term(s) containing `x**n`. If `n` is zero then all terms independent of `x` will be returned.

See also:

as_coefficient
separate the expression into a coefficient and factor

as_coeff_Add
separate the additive constant from an expression

as_coeff_Mul
separate the multiplicative constant from an expression

as_independent
separate x-dependent terms/factors from others

sympy.polys.polytools.Poly.coeff_monomial
efficiently find the single coefficient of a monomial in Poly

sympy.polys.polytools.Poly.nth
like coeff_monomial but powers of monomial terms are used

Examples

```
>>> from sympy import symbols
>>> from sympy.abc import x, y, z
```

You can select terms that have an explicit negative in front of them:

```
>>> (-x + 2*y).coeff(-1)
x
>>> (x - 2*y).coeff(-1)
2*y
```

You can select terms with no Rational coefficient:

```
>>> (x + 2*y).coeff(1)
x
>>> (3 + 2*x + 4*x**2).coeff(1)
0
```

You can select terms independent of x by making n=0; in this case expr.as_independent(x)[0] is returned (and 0 will be returned instead of None):

```
>>> (3 + 2*x + 4*x**2).coeff(x, 0)
3
>>> eq = ((x + 1)**3).expand() + 1
>>> eq
x**3 + 3*x**2 + 3*x + 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 2]
>>> eq -= 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 0]
```

You can select terms that have a numerical term in front of them:

```
>>> (-x - 2*y).coeff(2)
-y
>>> from sympy import sqrt
>>> (x + sqrt(2)*x).coeff(sqrt(2))
x
```

The matching is exact:

```
>>> (3 + 2*x + 4*x**2).coeff(x)
2
>>> (3 + 2*x + 4*x**2).coeff(x**2)
4
>>> (3 + 2*x + 4*x**2).coeff(x**3)
0
>>> (z*(x + y)**2).coeff((x + y)**2)
z
>>> (z*(x + y)**2).coeff(x + y)
0
```

In addition, no factoring is done, so 1 + z*(1 + y) is not obtained from the following:

```
>>> (x + z*(x + x*y)).coeff(x)
1
```

If such factoring is desired, factor_terms can be used first:

```
>>> from sympy import factor_terms
>>> factor_terms(x + z*(x + x*y)).coeff(x)
z*(y + 1) + 1
```

```
>>> n, m, o = symbols('n m o', commutative=False)
>>> n.coeff(n)
1
>>> (3*n).coeff(n)
3
>>> (n*m + m*n*m).coeff(n) # = (1 + m)*n*m
1 + m
>>> (n*m + m*n*m).coeff(n, right=True) # = (1 + m)*n*m
m
```

If there is more than one possible coefficient 0 is returned:

```
>>> (n*m + m*n).coeff(n)
0
```

If there is only one possible coefficient, it is returned:

```
>>> (n*m + x*m*n).coeff(m*n)
x
>>> (n*m + x*m*n).coeff(m*n, right=1)
1
```

collect(*syms*, *func=None*, *evaluate=True*, *exact=False*, *distribute_order_term=True*)

See the collect function in sympy.simplify

combsimp()

See the combsimp function in sympy.simplify

compare(*other*)

Return -1, 0, 1 if the object is smaller, equal, or greater than other.

Not in the mathematical sense. If the object is of a different type from the "other" then their classes are ordered according to the sorted_classes list.

Examples

```
>>> from sympy.abc import x, y
>>> x.compare(y)
-1
>>> x.compare(x)
0
>>> y.compare(x)
1
```

compute_leading_term(*x*, *logx=None*)

Deprecated function to compute the leading term of a series.

as_leading_term is only allowed for results of .series() This is a wrapper to compute a series first.

conjugate()

Returns the complex conjugate of 'self'.

copy()

could_extract_minus_sign()

Return True if self has -1 as a leading factor or has more literal negative signs than positive signs in a sum, otherwise False.

Examples

```
>>> from sympy.abc import x, y
>>> e = x - y
>>> {i.could_extract_minus_sign() for i in (e, -e)}
{False, True}
```

Though the `y - x` is considered like `-(x - y)`, since it is in a product without a leading factor of -1, the result is false below:

```
>>> (x*(y - x)).could_extract_minus_sign()
False
```

To put something in canonical form wrt to sign, use *signsimp*:

```
>>> from sympy import signsimp
>>> signsimp(x*(y - x))
-x*(x - y)
>>> _.could_extract_minus_sign()
True
```

count(*query*)

Count the number of matching subexpressions.

count_ops(*visual=None*)

Wrapper for count_ops that returns the operation count.

default_assumptions = {}

diff(**symbols*, ***assumptions*)

dir(*x*, *cdir*)

doit(***hints*)

Evaluate objects that are not evaluated by default like limits, integrals, sums and products. All objects of this kind will be evaluated recursively, unless some species were excluded via 'hints' or unless the 'deep' hint was set to 'False'.

```
>>> from sympy import Integral
>>> from sympy.abc import x
```

```
>>> 2*Integral(x, x)
2*Integral(x, x)
```

```
>>> (2*Integral(x, x)).doit()
x**2
```

```
>>> (2*Integral(x, x)).doit(deep=False)
2*Integral(x, x)
```

dummy_eq(*other*, *symbol=None*)

Compare two expressions and handle dummy symbols.

Examples

```
>>> from sympy import Dummy
>>> from sympy.abc import x, y
```

```
>>> u = Dummy('u')
```

```
>>> (u**2 + 1).dummy_eq(x**2 + 1)
True
>>> (u**2 + 1) == (x**2 + 1)
False
```

```
>>> (u**2 + y).dummy_eq(x**2 + y, x)
True
>>> (u**2 + y).dummy_eq(x**2 + y, y)
False
```

dummy_index

equals(*other*, *failing_expression=False*)

Return True if self == other, False if it does not, or None. If failing_expression is True then the expression which did not simplify to a 0 will be returned instead of None.

evalf(*n=15*, *subs=None*, *maxn=100*, *chop=False*, *strict=False*, *quad=None*, *verbose=False*)

Evaluate the given formula to an accuracy of *n* digits.

Parameters

subs
[dict, optional] Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

maxn
[int, optional] Allow a maximum temporary working precision of maxn digits.

chop
[bool or number, optional] Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When `True` the chop value defaults to standard precision.

Otherwise the chop value is used to determine the magnitude of "small" for purposes of chopping.

```
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
0.000100000000000000
>>> N(x, chop=1e-5)
0.000100000000000000
>>> N(x, chop=1e-4)
0
```

strict
[bool, optional] Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available maxprec.

quad
[str, optional] Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

verbose
[bool, optional] Print debug information.

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

`expand`(*deep=True*, *modulus=None*, *power_base=True*, *power_exp=True*, *mul=True*, *log=True*, *multinomial=True*, *basic=True*, ***hints*)

Expand an expression using hints.

See the docstring of the expand() function in sympy.core.function for more information.

`property expr_free_symbols`

Like `free_symbols`, but returns the free symbols only if they are contained in an expression node.

Examples

```
>>> from sympy.abc import x, y
>>> (x + y).expr_free_symbols
{x, y}
```

If the expression is contained in a non-expression object, do not return the free symbols. Compare:

```
>>> from sympy import Tuple
>>> t = Tuple(x + y)
>>> t.expr_free_symbols
set()
>>> t.free_symbols
{x, y}
```

extract_additively(*c*)

Return self - c if it's possible to subtract c from self and make all matching coefficients move towards zero, else return None.

See also:

extract_multiplicatively
coeff
as_coefficient

Examples

```
>>> from sympy.abc import x, y
>>> e = 2*x + 3
>>> e.extract_additively(x + 1)
x + 2
>>> e.extract_additively(3*x)
>>> e.extract_additively(4)
>>> (y*(x + 1)).extract_additively(x + 1)
>>> ((x + 1)*(x + 2*y + 1) + 3).extract_additively(x + 1)
(x + 1)*(x + 2*y) + 3
```

extract_branch_factor(*allow_half=False*)

Try to write self as `exp_polar(2*pi*I*n)*z` in a nice way. Return (z, n).

```
>>> from sympy import exp_polar, I, pi
>>> from sympy.abc import x, y
>>> exp_polar(I*pi).extract_branch_factor()
(exp_polar(I*pi), 0)
>>> exp_polar(2*I*pi).extract_branch_factor()
(1, 1)
>>> exp_polar(-pi*I).extract_branch_factor()
(exp_polar(I*pi), -1)
>>> exp_polar(3*pi*I + x).extract_branch_factor()
(exp_polar(x + I*pi), 1)
>>> (y*exp_polar(-5*pi*I)*exp_polar(3*pi*I + 2*pi*x)).extract_branch_factor()
(y*exp_polar(2*pi*x), -1)
```

(continues on next page)

(continued from previous page)

```
>>> exp_polar(-I*pi/2).extract_branch_factor()
(exp_polar(-I*pi/2), 0)
```

If allow_half is True, also extract exp_polar(I*pi):

```
>>> exp_polar(I*pi).extract_branch_factor(allow_half=True)
(1, 1/2)
>>> exp_polar(2*I*pi).extract_branch_factor(allow_half=True)
(1, 1)
>>> exp_polar(3*I*pi).extract_branch_factor(allow_half=True)
(1, 3/2)
>>> exp_polar(-I*pi).extract_branch_factor(allow_half=True)
(1, -1/2)
```

extract_multiplicatively(*c*)

Return None if it's not possible to make self in the form c * something in a nice way, i.e. preserving the properties of arguments of self.

Examples

```
>>> from sympy import symbols, Rational
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> ((x*y)**3).extract_multiplicatively(x**2 * y)
x*y**2
```

```
>>> ((x*y)**3).extract_multiplicatively(x**4 * y)
```

```
>>> (2*x).extract_multiplicatively(2)
x
```

```
>>> (2*x).extract_multiplicatively(3)
```

```
>>> (Rational(1, 2)*x).extract_multiplicatively(3)
x/6
```

factor(**gens*, ***args*)

See the factor() function in sympy.polys.polytools

find(*query*, *group=False*)

Find all subexpressions matching a query.

fourier_series(*limits=None*)

Compute fourier sine/cosine series of self.

See the docstring of the *fourier_series()* in sympy.series.fourier for more information.

fps(*x=None*, *x0=0*, *dir=1*, *hyper=True*, *order=4*, *rational=True*, *full=False*)

Compute formal power power series of self.

See the docstring of the *fps()* function in sympy.series.formal for more information.

property free_symbols

Return from the atoms of self those which are free symbols.

Not all free symbols are `Symbol`. Eg: IndexedBase('I')[0].free_symbols

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

classmethod fromiter(*args*, ***assumptions*)

Create a new object from an iterable.

This is a convenience function that allows one to create objects from any iterable, without having to convert to a list or tuple first.

Examples

```
>>> from sympy import Tuple
>>> Tuple.fromiter(i for i in range(5))
(0, 1, 2, 3, 4)
```

property func

The top-level function in an expression.

The following should hold for all objects:

```
>> x == x.func(*x.args)
```

Examples

```
>>> from sympy.abc import x
>>> a = 2*x
>>> a.func
<class 'sympy.core.mul.Mul'>
>>> a.args
(2, x)
>>> a.func(*a.args)
2*x
>>> a == a.func(*a.args)
True
```

gammasimp()

See the gammasimp function in sympy.simplify

getO()

Returns the additive O(..) symbol if there is one, else None.

getn()

Returns the order of the expression.

Examples

```
>>> from sympy import O
>>> from sympy.abc import x
>>> (1 + x + O(x**2)).getn()
2
>>> (1 + x).getn()
```

has(**patterns*)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y, z
>>> (x**2 + sin(x*y)).has(z)
False
>>> (x**2 + sin(x*y)).has(x, y, z)
True
>>> x.has(x)
True
```

Note `has` is a structural algorithm with no knowledge of mathematics. Consider the following half-open interval:

```
>>> from sympy import Interval
>>> i = Interval.Lopen(0, 5); i
Interval.Lopen(0, 5)
>>> i.args
(0, 5, True, False)
>>> i.has(4)  # there is no "4" in the arguments
False
>>> i.has(0)  # there *is* a "0" in the arguments
True
```

Instead, use `contains` to determine whether a number is in the interval or not:

```
>>> i.contains(4)
True
>>> i.contains(0)
False
```

Note that `expr.has(*patterns)` is exactly equivalent to `any(expr.has(p) for p in patterns)`. In particular, `False` is returned when the list of patterns is empty.

```
>>> x.has()
False
```

has_free(**patterns*)

Return True if self has object(s) `x` as a free expression else False.

Examples

```
>>> from sympy import Integral, Function
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> g = Function('g')
>>> expr = Integral(f(x), (f(x), 1, g(y)))
>>> expr.free_symbols
{y}
>>> expr.has_free(g(y))
True
>>> expr.has_free(*(x, f(x)))
False
```

This works for subexpressions and types, too:

```
>>> expr.has_free(g)
True
>>> (x + y + 1).has_free(y + 1)
True
```

has_xfree(*s: set[Basic]*)

Return True if self has any of the patterns in s as a free argument, else False. This is like *Basic.has_free* but this will only report exact argument matches.

Examples

```
>>> from sympy import Function
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> f(x).has_xfree({f})
False
>>> f(x).has_xfree({f(x)})
True
>>> f(x + 1).has_xfree({x})
True
>>> f(x + 1).has_xfree({x + 1})
True
>>> f(x + y + 1).has_xfree({x + 1})
False
```

integrate(**args*, ***kwargs*)

See the integrate function in sympy.integrals

invert(*g*, **gens*, ***args*)

Return the multiplicative inverse of `self` mod g where `self` (and g) may be symbolic expressions).

See also:

`sympy.core.numbers.mod_inverse`, `sympy.polys.polytools.invert`

is_Add = False

is_AlgebraicNumber = False

is_Atom = True

is_Boolean = False

is_Derivative = False

is_Dummy = True

is_Equality = False

is_Float = False

is_Function = False

is_Indexed = False

is_Integer = False

is_MatAdd = False

is_MatMul = False

is_Matrix = False

is_Mul = False

is_Not = False

is_Number = False

is_NumberSymbol = False

is_Order = False

is_Piecewise = False

is_Point = False

is_Poly = False

is_Pow = False

is_Rational = False

is_Relational = False

is_Symbol = True

is_Vector = False

is_Wild = False

property is_algebraic

is_algebraic_expr(**syms*)

This tests whether a given expression is algebraic or not, in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are "algebraic expressions" with symbolic exponents. This is a simple extension to the is_rational_function, including rational exponentiation.

See also:

is_rational_function

References

[1]

Examples

```
>>> from sympy import Symbol, sqrt
>>> x = Symbol('x', real=True)
>>> sqrt(1 + x).is_rational_function()
False
>>> sqrt(1 + x).is_algebraic_expr()
True
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be an algebraic expression to become one.

```
>>> from sympy import exp, factor
>>> a = sqrt(exp(x)**2 + 2*exp(x) + 1)/(exp(x) + 1)
>>> a.is_algebraic_expr(x)
False
>>> factor(a).is_algebraic_expr()
True
```

property is_antihermitian

property is_commutative

is_comparable = False

property is_complex

property is_composite

is_constant(**wrt*, ***flags*)

Return True if self is constant, False if not, or None if the constancy could not be determined conclusively.

Examples

```
>>> from sympy import cos, sin, Sum, S, pi
>>> from sympy.abc import a, n, x, y
>>> x.is_constant()
False
>>> S(2).is_constant()
True
>>> Sum(x, (x, 1, 10)).is_constant()
True
>>> Sum(x, (x, 1, n)).is_constant()
False
>>> Sum(x, (x, 1, n)).is_constant(y)
True
>>> Sum(x, (x, 1, n)).is_constant(n)
False
>>> Sum(x, (x, 1, n)).is_constant(x)
True
>>> eq = a*cos(x)**2 + a*sin(x)**2 - a
>>> eq.is_constant()
True
>>> eq.subs({x: pi, a: 2}) == eq.subs({x: pi, a: 3}) == 0
True
```

```
>>> (0**x).is_constant()
False
>>> x.is_constant()
False
>>> (x**x).is_constant()
False
>>> one = cos(x)**2 + sin(x)**2
>>> one.is_constant()
True
>>> ((one - 1)**(x + 1)).is_constant() in (True, False) # could be 0 or 1
True
```

property is_even

property is_extended_negative

property is_extended_nonnegative

property is_extended_nonpositive

property is_extended_nonzero

property is_extended_positive

property is_extended_real

property is_finite

property is_hermitian

is_hypergeometric(*k*)

property is_imaginary

property is_infinite

property is_integer

property is_irrational

is_meromorphic(*x*, *a*)

This tests whether an expression is meromorphic as a function of the given symbol `x` at the point `a`.

This method is intended as a quick test that will return None if no decision can be made without simplification or more detailed analysis.

Examples

```
>>> from sympy import zoo, log, sin, sqrt
>>> from sympy.abc import x
```

```
>>> f = 1/x**2 + 1 - 2*x**3
>>> f.is_meromorphic(x, 0)
True
>>> f.is_meromorphic(x, 1)
True
>>> f.is_meromorphic(x, zoo)
True
```

```
>>> g = x**log(3)
>>> g.is_meromorphic(x, 0)
False
>>> g.is_meromorphic(x, 1)
True
>>> g.is_meromorphic(x, zoo)
False
```

```
>>> h = sin(1/x)*x**2
>>> h.is_meromorphic(x, 0)
False
>>> h.is_meromorphic(x, 1)
True
>>> h.is_meromorphic(x, zoo)
True
```

Multivalued functions are considered meromorphic when their branches are meromorphic. Thus most functions are meromorphic everywhere except at essential singularities and branch points. In particular, they will be meromorphic also on branch cuts except at their endpoints.

```
>>> log(x).is_meromorphic(x, -1)
True
>>> log(x).is_meromorphic(x, 0)
False
>>> sqrt(x).is_meromorphic(x, -1)
True
```

(continues on next page)

(continued from previous page)

```
>>> sqrt(x).is_meromorphic(x, 0)
False
```

property is_negative

property is_noninteger

property is_nonnegative

property is_nonpositive

property is_nonzero

is_number = False

property is_odd

property is_polar

is_polynomial(**syms*)

Return True if self is a polynomial in syms and False otherwise.

This checks if self is an exact polynomial in syms. This function returns False for expressions that are "polynomials" with symbolic exponents. Thus, you should be able to apply polynomial algorithms to expressions for which this returns True, and Poly(expr, *syms) should work if and only if expr.is_polynomial(*syms) returns True. The polynomial does not have to be in expanded form. If no symbols are given, all free symbols in the expression will be used.

This is not part of the assumptions system. You cannot do Symbol('z', polynomial=True).

Examples

```
>>> from sympy import Symbol, Function
>>> x = Symbol('x')
>>> ((x**2 + 1)**4).is_polynomial(x)
True
>>> ((x**2 + 1)**4).is_polynomial()
True
>>> (2**x + 1).is_polynomial(x)
False
>>> (2**x + 1).is_polynomial(2**x)
True
>>> f = Function('f')
>>> (f(x) + 1).is_polynomial(x)
False
>>> (f(x) + 1).is_polynomial(f(x))
True
>>> (1/f(x) + 1).is_polynomial(f(x))
False
```

```
>>> n = Symbol('n', nonnegative=True, integer=True)
>>> (x**n + 1).is_polynomial(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a polynomial to become one.

```
>>> from sympy import sqrt, factor, cancel
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)
>>> a.is_polynomial(y)
False
>>> factor(a)
y + 1
>>> factor(a).is_polynomial(y)
True
```

```
>>> b = (y**2 + 2*y + 1)/(y + 1)
>>> b.is_polynomial(y)
False
>>> cancel(b)
y + 1
>>> cancel(b).is_polynomial(y)
True
```

See also .is_rational_function()

property is_positive

property is_prime

property is_rational

is_rational_function(**syms*)

Test whether function is a ratio of two polynomials in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are "rational functions" with symbolic exponents. Thus, you should be able to call .as_numer_denom() and apply polynomial algorithms to the result for expressions for which this returns True.

This is not part of the assumptions system. You cannot do Symbol('z', rational_function=True).

Examples

```
>>> from sympy import Symbol, sin
>>> from sympy.abc import x, y
```

```
>>> (x/y).is_rational_function()
True
```

```
>>> (x**2).is_rational_function()
True
```

```
>>> (x/sin(y)).is_rational_function(y)
False
```

```
>>> n = Symbol('n', integer=True)
>>> (x**n + 1).is_rational_function(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a rational function to become one.

```
>>> from sympy import sqrt, factor
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)/y
>>> a.is_rational_function(y)
False
>>> factor(a)
(y + 1)/y
>>> factor(a).is_rational_function(y)
True
```

See also is_algebraic_expr().

property is_real

is_scalar = True

is_symbol = True

property is_transcendental

property is_zero

property kind

Default kind for all SymPy object. If the kind is not defined for the object, or if the object cannot infer the kind from its arguments, this will be returned.

Examples

```
>>> from sympy import Expr
>>> Expr().kind
UndefinedKind
```

leadterm(*x*, *logx=None*, *cdir=0*)

Returns the leading term a*x**b as a tuple (a, b).

Examples

```
>>> from sympy.abc import x
>>> (1+x+x**2).leadterm(x)
(1, 0)
>>> (1/x**2+x+x**2).leadterm(x)
(1, -2)
```

limit(*x*, *xlim*, *dir='+'*)

Compute limit x->xlim.

lseries(*x=None*, *x0=0*, *dir='+'*, *logx=None*, *cdir=0*)

Wrapper for series yielding an iterator of the terms of the series.

Note: an infinite series will yield an infinite iterator. The following, for exaxmple, will never terminate. It will just keep printing terms of the sin(x) series:

```
for term in sin(x).lseries(x):
    print term
```

The advantage of lseries() over nseries() is that many times you are just interested in the next term in the series (i.e. the first term for example), but you do not know how many you should ask for in nseries() using the "n" parameter.

See also nseries().

match(*pattern*, *old=False*)

Pattern matching.

Wild symbols match all.

Return `None` when expression (self) does not match with pattern. Otherwise return a dictionary such that:

```
pattern.xreplace(self.match(pattern)) == self
```

Examples

```
>>> from sympy import Wild, Sum
>>> from sympy.abc import x, y
>>> p = Wild("p")
>>> q = Wild("q")
>>> r = Wild("r")
>>> e = (x+y)**(x+y)
>>> e.match(p**p)
{p_: x + y}
>>> e.match(p**q)
{p_: x + y, q_: x + y}
>>> e = (2*x)**2
>>> e.match(p*q**r)
{p_: 4, q_: x, r_: 2}
>>> (p*q**r).xreplace(e.match(p*q**r))
4*x**2
```

Structurally bound symbols are ignored during matching:

```
>>> Sum(x, (x, 1, 2)).match(Sum(y, (y, 1, p)))
{p_: 2}
```

But they can be identified if desired:

```
>>> Sum(x, (x, 1, 2)).match(Sum(q, (q, 1, p)))
{p_: 2, q_: x}
```

The `old` flag will give the old-style pattern matching where expressions and patterns are essentially solved to give the match. Both of the following give None unless `old=True`:

```
>>> (x - 2).match(p - x, old=True)
{p_: 2*x - 2}
>>> (2/x).match(p*x, old=True)
{p_: 2/x**2}
```

matches(*expr*, *repl_dict=None*, *old=False*)

Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.

Examples

```
>>> from sympy import symbols, Wild, Basic
>>> a, b, c = symbols('a b c')
>>> x = Wild('x')
>>> Basic(a + x, x).matches(Basic(a + b, c)) is None
True
>>> Basic(a + x, x).matches(Basic(a + b + c, b + c))
{x_: b + c}
```

n(*n=15*, *subs=None*, *maxn=100*, *chop=False*, *strict=False*, *quad=None*, *verbose=False*)

Evaluate the given formula to an accuracy of *n* digits.

Parameters

subs
[dict, optional] Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

maxn
[int, optional] Allow a maximum temporary working precision of maxn digits.

chop
[bool or number, optional] Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When `True` the chop value defaults to standard precision.

Otherwise the chop value is used to determine the magnitude of "small" for purposes of chopping.

```
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
0.000100000000000000
>>> N(x, chop=1e-5)
0.000100000000000000
>>> N(x, chop=1e-4)
0
```

strict
[bool, optional] Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available maxprec.

quad
[str, optional] Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

verbose
[bool, optional] Print debug information.

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

name: str

normal()

Return the expression as a fraction.

expression -> a/b

See also:

as_numer_denom
return (a, b) instead of a/b

nseries(*x=None*, *x0=0*, *n=6*, *dir='+'*, *logx=None*, *cdir=0*)

Wrapper to _eval_nseries if assumptions allow, else to series.

If x is given, x0 is 0, dir='+', and self has x, then _eval_nseries is called. This calculates "n" terms in the innermost expressions and then builds up the final series just by "cross-multiplying" everything out.

The optional `logx` parameter can be used to replace any log(x) in the returned series with a symbolic value to avoid evaluating log(x) at 0. A symbol to use in place of log(x) should be provided.

Advantage – it's fast, because we do not have to determine how many terms we need to calculate in advance.

Disadvantage – you may end up with less terms than you may have expected, but the O(x**n) term appended will always be correct and so the result, though perhaps shorter, will also be correct.

If any of those assumptions is not met, this is treated like a wrapper to series which will try harder to return the correct number of terms.

See also lseries().

Examples

```
>>> from sympy import sin, log, Symbol
>>> from sympy.abc import x, y
>>> sin(x).nseries(x, 0, 6)
x - x**3/6 + x**5/120 + O(x**6)
>>> log(x+1).nseries(x, 0, 5)
x - x**2/2 + x**3/3 - x**4/4 + O(x**5)
```

Handling of the `logx` parameter — in the following example the expansion fails since `sin` does not have an asymptotic expansion at -oo (the limit of log(x) as x approaches 0):

```
>>> e = sin(log(x))
>>> e.nseries(x, 0, 6)
Traceback (most recent call last):
...
PoleError: ...
...
>>> logx = Symbol('logx')
>>> e.nseries(x, 0, 6, logx=logx)
sin(logx)
```

In the following example, the expansion works but only returns self unless the `logx` parameter is used:

```
>>> e = x**y
>>> e.nseries(x, 0, 2)
x**y
>>> e.nseries(x, 0, 2, logx=logx)
exp(logx*y)
```

nsimplify(*constants=()*, *tolerance=None*, *full=False*)

See the nsimplify function in sympy.simplify

powsimp(**args*, ***kwargs*)

See the powsimp function in sympy.simplify

primitive()

Return the positive Rational that can be extracted non-recursively from every term of self (i.e., self is treated like an Add). This is like the as_coeff_Mul() method but primitive always extracts a positive Rational (never a negative or a Float).

Examples

```
>>> from sympy.abc import x
>>> (3*(x + 1)**2).primitive()
(3, (x + 1)**2)
>>> a = (6*x + 2); a.primitive()
(2, 3*x + 1)
>>> b = (x/2 + 3); b.primitive()
(1/2, x + 6)
>>> (a*b).primitive() == (1, a*b)
True
```

radsimp(***kwargs*)

See the radsimp function in sympy.simplify

ratsimp()

See the ratsimp function in sympy.simplify

rcall(**args*)

Apply on the argument recursively through the expression tree.

This method is used to simulate a common abuse of notation for operators. For instance, in SymPy the following will not work:

`(x+Lambda(y, 2*y))(z) == x+2*z`,

however, you can use:

```
>>> from sympy import Lambda
>>> from sympy.abc import x, y, z
>>> (x + Lambda(y, 2*y)).rcall(z)
x + 2*z
```

refine(*assumption=True*)

See the refine function in sympy.assumptions

removeO()

Removes the additive O(..) symbol if there is one

replace(*query*, *value*, *map=False*, *simultaneous=True*, *exact=None*)

Replace matching subexpressions of `self` with `value`.

If `map = True` then also return the mapping {old: new} where `old` was a sub-expression found with query and `new` is the replacement value for it. If the expression itself does not match the query, then the returned value will be `self.xreplace(map)` otherwise it should be `self.subs(ordered(map.items()))`.

Traverses an expression tree and performs replacement of matching subexpressions from the bottom to the top of the tree. The default approach is to do the replacement in a simultaneous fashion so changes made are targeted only once. If this is not desired or causes problems, `simultaneous` can be set to False.

In addition, if an expression containing more than one Wild symbol is being used to match subexpressions and the `exact` flag is None it will be set to True so the match will only succeed if all non-zero values are received for each Wild that appears in the match pattern. Setting this to False accepts a match of 0; while setting it True accepts all matches that have a 0 in them. See example below for cautions.

The list of possible combinations of queries and replacement values is listed below:

See also:

subs
substitution of subexpressions as defined by the objects themselves.

xreplace
exact node replacement in expr tree; also capable of using matching rules

Examples

Initial setup

```
>>> from sympy import log, sin, cos, tan, Wild, Mul, Add
>>> from sympy.abc import x, y
>>> f = log(sin(x)) + tan(sin(x**2))
```

1.1. type -> type
obj.replace(type, newtype)

When object of type `type` is found, replace it with the result of passing its argument(s) to `newtype`.

```
>>> f.replace(sin, cos)
log(cos(x)) + tan(cos(x**2))
>>> sin(x).replace(sin, cos, map=True)
(cos(x), {sin(x): cos(x)})
>>> (x*y).replace(Mul, Add)
x + y
```

1.2. type -> func
obj.replace(type, func)

When object of type `type` is found, apply `func` to its argument(s). `func` must be written to handle the number of arguments of `type`.

```
>>> f.replace(sin, lambda arg: sin(2*arg))
log(sin(2*x)) + tan(sin(2*x**2))
>>> (x*y).replace(Mul, lambda *args: sin(2*Mul(*args)))
sin(2*x*y)
```

2.1. pattern -> expr
obj.replace(pattern(wild), expr(wild))

Replace subexpressions matching `pattern` with the expression written in terms of the Wild symbols in `pattern`.

```
>>> a, b = map(Wild, 'ab')
>>> f.replace(sin(a), tan(a))
log(tan(x)) + tan(tan(x**2))
>>> f.replace(sin(a), tan(a/2))
log(tan(x/2)) + tan(tan(x**2/2))
>>> f.replace(sin(a), a)
log(x) + tan(x**2)
>>> (x*y).replace(a*x, a)
y
```

Matching is exact by default when more than one Wild symbol is used: matching fails unless the match gives non-zero values for all Wild symbols:

```
>>> (2*x + y).replace(a*x + b, b - a)
y - 2
>>> (2*x).replace(a*x + b, b - a)
2*x
```

When set to False, the results may be non-intuitive:

```
>>> (2*x).replace(a*x + b, b - a, exact=False)
2/x
```

2.2. pattern -> func

obj.replace(pattern(wild), lambda wild: expr(wild))

All behavior is the same as in 2.1 but now a function in terms of pattern variables is used rather than an expression:

```
>>> f.replace(sin(a), lambda a: sin(2*a))
log(sin(2*x)) + tan(sin(2*x**2))
```

3.1. func -> func

obj.replace(filter, func)

Replace subexpression `e` with `func(e)` if `filter(e)` is True.

```
>>> g = 2*sin(x**3)
>>> g.replace(lambda expr: expr.is_Number, lambda expr: expr**2)
4*sin(x**9)
```

The expression itself is also targeted by the query but is done in such a fashion that changes are not made twice.

```
>>> e = x*(x*y + 1)
>>> e.replace(lambda x: x.is_Mul, lambda x: 2*x)
2*x*(2*x*y + 1)
```

When matching a single symbol, *exact* will default to True, but this may or may not be the behavior that is desired:

Here, we want *exact=False*:

```
>>> from sympy import Function
>>> f = Function('f')
>>> e = f(1) + f(0)
>>> q = f(a), lambda a: f(a + 1)
>>> e.replace(*q, exact=False)
f(1) + f(2)
>>> e.replace(*q, exact=True)
f(0) + f(2)
```

But here, the nature of matching makes selecting the right setting tricky:

```
>>> e = x**(1 + y)
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**-a, exact=False)
x
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**-a, exact=True)
x**(-x - y + 1)
>>> (x**y).replace(x**(1 + a), lambda a: x**-a, exact=False)
x
>>> (x**y).replace(x**(1 + a), lambda a: x**-a, exact=True)
x**(1 - y)
```

It is probably better to use a different form of the query that describes the target expression more precisely:

```
>>> (1 + x**(1 + y)).replace(
... lambda x: x.is_Pow and x.exp.is_Add and x.exp.args[0] == 1,
... lambda x: x.base**(1 - (x.exp - 1)))
...
x**(1 - y) + 1
```

rewrite(**args*, *deep=True*, ***hints*)

Rewrite *self* using a defined rule.

Rewriting transforms an expression to another, which is mathematically equivalent but structurally different. For example you can rewrite trigonometric functions as complex exponentials or combinatorial functions as gamma function.

This method takes a *pattern* and a *rule* as positional arguments. *pattern* is optional parameter which defines the types of expressions that will be transformed. If it is not passed, all possible expressions will be rewritten. *rule* defines how the expression will be rewritten.

Parameters

args
[Expr] A *rule*, or *pattern* and *rule*. - *pattern* is a type or an iterable of types. - *rule* can be any object.

deep
[bool, optional] If `True`, subexpressions are recursively transformed. Default is `True`.

Examples

If *pattern* is unspecified, all possible expressions are transformed.

```
>>> from sympy import cos, sin, exp, I
>>> from sympy.abc import x
>>> expr = cos(x) + I*sin(x)
>>> expr.rewrite(exp)
exp(I*x)
```

Pattern can be a type or an iterable of types.

```
>>> expr.rewrite(sin, exp)
exp(I*x)/2 + cos(x) - exp(-I*x)/2
>>> expr.rewrite([cos,], exp)
exp(I*x)/2 + I*sin(x) + exp(-I*x)/2
>>> expr.rewrite([cos, sin], exp)
exp(I*x)
```

Rewriting behavior can be implemented by defining `_eval_rewrite()` method.

```
>>> from sympy import Expr, sqrt, pi
>>> class MySin(Expr):
...     def _eval_rewrite(self, rule, args, **hints):
...         x, = args
...         if rule == cos:
...             return cos(pi/2 - x, evaluate=False)
...         if rule == sqrt:
...             return sqrt(1 - cos(x)**2)
```

(continues on next page)

(continued from previous page)

```
>>> MySin(MySin(x)).rewrite(cos)
cos(-cos(-x + pi/2) + pi/2)
>>> MySin(x).rewrite(sqrt)
sqrt(1 - cos(x)**2)
```

Defining _eval_rewrite_as_[...]() method is supported for backwards compatibility reason. This may be removed in the future and using it is discouraged.

```
>>> class MySin(Expr):
...     def _eval_rewrite_as_cos(self, *args, **hints):
...         x, = args
...         return cos(pi/2 - x, evaluate=False)
>>> MySin(x).rewrite(cos)
cos(-x + pi/2)
```

round(*n=None*)

Return x rounded to the given decimal place.

If a complex number would results, apply round to the real and imaginary components of the number.

Notes

The Python round function uses the SymPy round method so it will always return a SymPy number (not a Python float or int):

```
>>> isinstance(round(S(123), -2), Number)
True
```

Examples

```
>>> from sympy import pi, E, I, S, Number
>>> pi.round()
3
>>> pi.round(2)
3.14
>>> (2*pi + E*I).round()
6 + 3*I
```

The round method has a chopping effect:

```
>>> (2*pi + I/10).round()
6
>>> (pi/10 + 2*I).round()
2*I
>>> (pi/10 + E*I).round(2)
0.31 + 2.72*I
```

separate(*deep=False*, *force=False*)

See the separate function in sympy.simplify

series(*x=None*, *x0=0*, *n=6*, *dir='+'*, *logx=None*, *cdir=0*)

Series expansion of "self" around `x = x0` yielding either terms of the series one by one (the lazy series given when n=None), else all the terms at once when n != None.

Returns the series expansion of "self" around the point `x = x0` with respect to `x` up to `O((x - x0)**n, x, x0)` (default n is 6).

If `x=None` and `self` is univariate, the univariate symbol will be supplied, otherwise an error will be raised.

Parameters

expr
[Expression] The expression whose series is to be expanded.

x
[Symbol] It is the variable of the expression to be calculated.

x0
[Value] The value around which `x` is calculated. Can be any value from `-oo` to `oo`.

n
[Value] The value used to represent the order in terms of `x**n`, up to which the series is to be expanded.

dir
[String, optional] The series-expansion can be bi-directional. If `dir="+"`, then (x->x0+). If `dir="-", then (x->x0-). For infinite ``x0` (`oo` or `-oo`), the `dir` argument is determined from the direction of the infinity (i.e., `dir="-"` for `oo`).

logx
[optional] It is used to replace any log(x) in the returned series with a symbolic value rather than evaluating the actual value.

cdir
[optional] It stands for complex direction, and indicates the direction from which the expansion needs to be evaluated.

Returns

Expr
[Expression] Series expansion of the expression about x0

Raises

TypeError
If "n" and "x0" are infinity objects

PoleError
If "x0" is an infinity object

Examples

```
>>> from sympy import cos, exp, tan
>>> from sympy.abc import x, y
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
>>> cos(x).series(n=4)
1 - x**2/2 + O(x**4)
>>> cos(x).series(x, x0=1, n=2)
```

(continues on next page)

(continued from previous page)

```
cos(1) - (x - 1)*sin(1) + O((x - 1)**2, (x, 1))
>>> e = cos(x + exp(y))
>>> e.series(y, n=2)
cos(x + 1) - y*sin(x + 1) + O(y**2)
>>> e.series(x, n=2)
cos(exp(y)) - x*sin(exp(y)) + O(x**2)
```

If n=None then a generator of the series terms will be returned.

```
>>> term=cos(x).series(n=None)
>>> [next(term) for i in range(2)]
[1, -x**2/2]
```

For dir=+ (default) the series is calculated from the right and for dir=- the series from the left. For smooth functions this flag will not alter the results.

```
>>> abs(x).series(dir="+")
x
>>> abs(x).series(dir="-")
-x
>>> f = tan(x)
>>> f.series(x, 2, 6, "+")
tan(2) + (1 + tan(2)**2)*(x - 2) + (x - 2)**2*(tan(2)**3 + tan(2)) +
(x - 2)**3*(1/3 + 4*tan(2)**2/3 + tan(2)**4) + (x - 2)**4*(tan(2)**5 +
5*tan(2)**3/3 + 2*tan(2)/3) + (x - 2)**5*(2/15 + 17*tan(2)**2/15 +
2*tan(2)**4 + tan(2)**6) + O((x - 2)**6, (x, 2))
```

```
>>> f.series(x, 2, 3, "-")
tan(2) + (2 - x)*(-tan(2)**2 - 1) + (2 - x)**2*(tan(2)**3 + tan(2))
+ O((x - 2)**3, (x, 2))
```

For rational expressions this method may return original expression without the Order term. >>> (1/x).series(x, n=8) 1/x

simplify(***kwargs*)

See the simplify function in sympy.simplify

sort_key(*order=None*)

Return a sort key.

Examples

```
>>> from sympy import S, I
```

```
>>> sorted([S(1)/2, I, -I], key=lambda x: x.sort_key())
[1/2, -I, I]
```

```
>>> S("[x, 1/x, 1/x**2, x**2, x**(1/2), x**(1/4), x**(3/2)]")
[x, 1/x, x**(-2), x**2, sqrt(x), x**(1/4), x**(3/2)]
>>> sorted(_, key=lambda x: x.sort_key())
[x**(-2), 1/x, x**(1/4), sqrt(x), x, x**(3/2), x**2]
```

subs(**args*, ***kwargs*)

Substitutes old for new in an expression after sympifying args.

***args* is either:**

- two arguments, e.g. foo.subs(old, new)
- **one iterable argument, e.g. foo.subs(iterable). The iterable may be**

  o an iterable container with (old, new) pairs. In this case the
  replacements are processed in the order given with successive patterns possibly affecting replacements already made.

  o a dict or set whose key/value items correspond to old/new pairs.
  In this case the old/new pairs will be sorted by op count and in case of a tie, by number of args and the default_sort_key. The resulting sorted list is then processed as an iterable container (see previous).

If the keyword `simultaneous` is True, the subexpressions will not be evaluated until all the substitutions have been made.

See also:

replace
replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

xreplace
exact node replacement in expr tree; also capable of using matching rules

`sympy.core.evalf.EvalfMixin.evalf`
calculates the given formula to a desired level of precision

Examples

```
>>> from sympy import pi, exp, limit, oo
>>> from sympy.abc import x, y
>>> (1 + x*y).subs(x, pi)
pi*y + 1
>>> (1 + x*y).subs({x:pi, y:2})
1 + 2*pi
>>> (1 + x*y).subs([(x, pi), (y, 2)])
1 + 2*pi
>>> reps = [(y, x**2), (x, 2)]
>>> (x + y).subs(reps)
6
>>> (x + y).subs(reversed(reps))
x**2 + 2
```

```
>>> (x**2 + x**4).subs(x**2, y)
y**2 + y
```

To replace only the x**2 but not the x**4, use xreplace:

```
>>> (x**2 + x**4).xreplace({x**2: y})
x**4 + y
```

To delay evaluation until all substitutions have been made, set the keyword `simultaneous` to True:

```
>>> (x/y).subs([(x, 0), (y, 0)])
0
>>> (x/y).subs([(x, 0), (y, 0)], simultaneous=True)
nan
```

This has the added feature of not allowing subsequent substitutions to affect those already made:

```
>>> ((x + y)/y).subs({x + y: y, y: x + y})
1
>>> ((x + y)/y).subs({x + y: y, y: x + y}, simultaneous=True)
y/(x + y)
```

In order to obtain a canonical result, unordered iterables are sorted by count_op length, number of arguments and by the default_sort_key to break any ties. All other iterables are left unsorted.

```
>>> from sympy import sqrt, sin, cos
>>> from sympy.abc import a, b, c, d, e
```

```
>>> A = (sqrt(sin(2*x)), a)
>>> B = (sin(2*x), b)
>>> C = (cos(2*x), c)
>>> D = (x, d)
>>> E = (exp(x), e)
```

```
>>> expr = sqrt(sin(2*x))*sin(exp(x)*x)*cos(2*x) + sin(2*x)
```

```
>>> expr.subs(dict([A, B, C, D, E]))
a*c*sin(d*e) + b
```

The resulting expression represents a literal replacement of the old arguments with the new arguments. This may not reflect the limiting behavior of the expression:

```
>>> (x**3 - 3*x).subs({x: oo})
nan
```

```
>>> limit(x**3 - 3*x, x, oo)
oo
```

If the substitution will be followed by numerical evaluation, it is better to pass the substitution to evalf as

```
>>> (1/x).evalf(subs={x: 3.0}, n=21)
0.333333333333333333333
```

rather than

```
>>> (1/x).subs({x: 3.0}).evalf(21)
0.333333333333333314830
```

as the former will ensure that the desired level of precision is obtained.

taylor_term(*n*, *x*, **previous_terms*)

General method for the taylor term.

This method is slow, because it differentiates n-times. Subclasses can redefine it to make it faster by using the "previous_terms".

to_nnf(*simplify=True*)

together(**args*, ***kwargs*)

See the together function in sympy.polys

transpose()

trigsimp(***args*)

See the trigsimp function in sympy.simplify

xreplace(*rule*, *hack2=False*)

Replace occurrences of objects within the expression.

Parameters

rule
[dict-like] Expresses a replacement rule

Returns

xreplace
[the result of the replacement]

See also:

replace
replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

subs
substitution of subexpressions as defined by the objects themselves.

Examples

```
>>> from sympy import symbols, pi, exp
>>> x, y, z = symbols('x y z')
>>> (1 + x*y).xreplace({x: pi})
pi*y + 1
>>> (1 + x*y).xreplace({x: pi, y: 2})
1 + 2*pi
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> (x*y + z).xreplace({x*y: pi})
z + pi
>>> (x*y*z).xreplace({x*y: pi})
x*y*z
>>> (2*x).xreplace({2*x: y, x: z})
y
>>> (2*2*x).xreplace({2*x: y, x: z})
4*z
>>> (x + y + 2).xreplace({x + y: 2})
x + y + 2
>>> (x + 2 + exp(x + 2)).xreplace({x + 2: y})
x + exp(y) + 2
```

xreplace does not differentiate between free and bound symbols. In the following, subs(x, y) would not change x since it is a bound symbol, but xreplace does:

```
>>> from sympy import Integral
>>> Integral(x, (x, 1, 2*x)).xreplace({x: y})
Integral(y, (y, 1, 2*y))
```

Trying to replace x with an expression raises an error:

```
>>> Integral(x, (x, 1, 2*x)).xreplace({x: 2*y})
ValueError: Invalid limits given: ((2*y, 1, 4*y),)
```

58.2.2 `Factor`

class `nipy.algorithms.statistics.formula.formulae.`**`Factor`**(*name*, *levels*, *char='b'*)

Bases: *`Formula`*

A qualitative variable in a regression model

A Factor is similar to R's factor. The levels of the Factor can be either strings or ints.

`__init__`(*name*, *levels*, *char='b'*)

Initialize Factor

Parameters

name
[str]

levels
[[str or int]] A sequence of strings or ints.

char
[str, optional] prefix character for regression coefficients

property coefs

Coefficients in the linear regression formula.

`design`(*input*, *param=None*, *return_float=False*, *contrasts=None*)

Construct the design matrix, and optional contrast matrices.

Parameters

input
[np.recarray] Recarray including fields needed to compute the Terms in getparams(self.design_expr).

param
[None or np.recarray] Recarray including fields that are not Terms in getparams(self.design_expr)

return_float
[bool, optional] If True, return a np.float64 array rather than a np.recarray

contrasts
[None or dict, optional] Contrasts. The items in this dictionary should be (str, Formula) pairs where a contrast matrix is constructed for each Formula by evaluating its design at the same parameters as self.design. If not None, then the return_float is set to True.

Returns

des
[2D array] design matrix

cmatrices
[dict, optional] Dictionary with keys from *contrasts* input, and contrast matrices corresponding to *des* design matrix. Returned only if *contrasts* input is not None

property design_expr

property dtype

The dtype of the design matrix of the Formula.

static fromcol(*col*, *name*)

Create a Factor from a column array.

Parameters

col
[ndarray] an array with ndim==1

name
[str] name of the Factor

Returns

factor
[Factor]

Examples

```
>>> data = np.array([(3,'a'),(4,'a'),(5,'b'),(3,'b')], np.dtype([('x', np.
↪float64), ('y', 'S1')]))
>>> f1 = Factor.fromcol(data['y'], 'y')
>>> f2 = Factor.fromcol(data['x'], 'x')
>>> d = f1.design(data)
>>> print(d.dtype.descr)
[('y_a', '<f8'), ('y_b', '<f8')]
>>> d = f2.design(data)
>>> print(d.dtype.descr)
[('x_3', '<f8'), ('x_4', '<f8'), ('x_5', '<f8')]
```

static fromrec(*rec*, *keep=[]*, *drop=[]*)

Construct Formula from recarray

For fields with a string-dtype, it is assumed that these are qualtiatitve regressors, i.e. Factors.

Parameters

rec: recarray
Recarray whose field names will be used to create a formula.

keep: []
Field names to explicitly keep, dropping all others.

drop: []
Field names to drop.

get_term(*level*)

Retrieve a term of the Factor...

property main_effect

property mean

Expression for the mean, expressed as a linear combination of terms, each with dummy variables in front.

property params

The parameters in the Formula.

stratify(*variable*)

Create a new variable, stratified by the levels of a Factor.

Parameters

variable
[str or simple sympy expression] If sympy expression, then string representation must be all lower or upper case letters, i.e. it can be interpreted as a name.

Returns

formula
[Formula] Formula whose mean has one parameter named variable%d, for each level in self.levels

Examples

```
>>> f = Factor('a', ['x','y'])
>>> sf = f.stratify('theta')
>>> sf.mean
_theta0*a_x + _theta1*a_y
```

subs(*old*, *new*)

Perform a sympy substitution on all terms in the Formula

Returns a new instance of the same class

Parameters

old
[sympy.Basic] The expression to be changed

new
[sympy.Basic] The value to change it to.

Returns

newf
[Formula]

Examples

```
>>> s, t = [Term(l) for l in 'st']
>>> f, g = [sympy.Function(l) for l in 'fg']
>>> form = Formula([f(t),g(s)])
>>> newform = form.subs(g, sympy.Function('h'))
>>> newform.terms
array([f(t), h(s)], dtype=object)
```

(continues on next page)

(continued from previous page)

```
>>> form.terms
array([f(t), g(s)], dtype=object)
```

property terms

Terms in the linear regression formula.

58.2.3 FactorTerm

class nipy.algorithms.statistics.formula.formulae.**FactorTerm**(*name*, *level*)

Bases: *Term*

Boolean Term derived from a Factor.

Its properties are the same as a Term except that its product with itself is itself.

__init__(**args*, ***kwargs*)

adjoint()

apart(*x=None*, ***args*)

See the apart function in sympy.polys

property args: tuple[Basic, ...]

Returns a tuple of arguments of 'self'.

Notes

Never use self._args, always use self.args. Only use _args in __new__ when creating a new function. Do not override .args() from Basic (so that it is easy to change the interface in the future if needed).

Examples

```
>>> from sympy import cot
>>> from sympy.abc import x, y
```

```
>>> cot(x).args
(x,)
```

```
>>> cot(x).args[0]
x
```

```
>>> (x*y).args
(x, y)
```

```
>>> (x*y).args[1]
y
```

args_cnc(*cset=False*, *warn=True*, *split_1=True*)

Return [commutative factors, non-commutative factors] of self.

Examples

```
>>> from sympy import symbols, oo
>>> A, B = symbols('A B', commutative=0)
>>> x, y = symbols('x y')
>>> (-2*x*y).args_cnc()
[[-1, 2, x, y], []]
>>> (-2.5*x).args_cnc()
[[-1, 2.5, x], []]
>>> (-2*x*A*B*y).args_cnc()
[[-1, 2, x, y], [A, B]]
>>> (-2*x*A*B*y).args_cnc(split_1=False)
[[-2, x, y], [A, B]]
>>> (-2*x*y).args_cnc(cset=True)
[{-1, 2, x, y}, []]
```

The arg is always treated as a Mul:

```
>>> (-2 + x + A).args_cnc()
[[], [x - 2 + A]]
>>> (-oo).args_cnc() # -oo is a singleton
[[-1, oo], []]
```

as_base_exp() → tuple[Expr, Expr]

as_coeff_Add(*rational=False*) → tuple[Number, Expr]

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational: bool = False*) → tuple[Number, Expr]

Efficiently extract the coefficient of a product.

as_coeff_add(**deps*) → tuple[Expr, tuple[Expr, ...]]

Return the tuple (c, args) where self is written as an Add, `a`.

c should be a Rational added to any terms of the Add that are independent of deps.

args should be a tuple of all other terms of `a`; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you do not know if self is an Add or not but you want to treat self as an Add or if you want to process the individual arguments of the tail of self as an Add.

- if you know self is an Add and want only the head, use self.args[0];
- if you do not want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail.
- if you want to split self into an independent and dependent parts use `self.as_independent(*deps)`

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_add()
(3, ())
>>> (3 + x).as_coeff_add()
(3, (x,))
>>> (3 + x + y).as_coeff_add(x)
(y + 3, (x,))
```

(continues on next page)

(continued from previous page)

```
>>> (3 + y).as_coeff_add(x)
(y + 3, ())
```

as_coeff_exponent(*x*) → tuple[Expr, Expr]

`c*x**e -> c,e` where x can be any symbolic expression.

as_coeff_mul(**deps*, ***kwargs*) → tuple[Expr, tuple[Expr, ...]]

Return the tuple (c, args) where self is written as a Mul, `m`.

c should be a Rational multiplied by any factors of the Mul that are independent of deps.

args should be a tuple of all other factors of m; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you do not know if self is a Mul or not but you want to treat self as a Mul or if you want to process the individual arguments of the tail of self as a Mul.

- if you know self is a Mul and want only the head, use self.args[0];
- if you do not want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail;
- if you want to split self into an independent and dependent parts use `self.as_independent(*deps)`

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_mul()
(3, ())
>>> (3*x*y).as_coeff_mul()
(3, (x, y))
>>> (3*x*y).as_coeff_mul(x)
(3*y, (x,))
>>> (3*y).as_coeff_mul(x)
(3*y, ())
```

as_coefficient(*expr*)

Extracts symbolic coefficient at the given expression. In other words, this functions separates 'self' into the product of 'expr' and 'expr'-free coefficient. If such separation is not possible it will return None.

See also:

coeff
return sum of terms have a given factor

as_coeff_Add
separate the additive constant from an expression

as_coeff_Mul
separate the multiplicative constant from an expression

as_independent
separate x-dependent terms/factors from others

sympy.polys.polytools.Poly.coeff_monomial
efficiently find the single coefficient of a monomial in Poly

sympy.polys.polytools.Poly.nth
like coeff_monomial but powers of monomial terms are used

Examples

```
>>> from sympy import E, pi, sin, I, Poly
>>> from sympy.abc import x
```

```
>>> E.as_coefficient(E)
1
>>> (2*E).as_coefficient(E)
2
>>> (2*sin(E)*E).as_coefficient(E)
```

Two terms have E in them so a sum is returned. (If one were desiring the coefficient of the term exactly matching E then the constant from the returned expression could be selected. Or, for greater precision, a method of Poly can be used to indicate the desired term from which the coefficient is desired.)

```
>>> (2*E + x*E).as_coefficient(E)
x + 2
>>> _.args[0]  # just want the exact match
2
>>> p = Poly(2*E + x*E); p
Poly(x*E + 2*E, x, E, domain='ZZ')
>>> p.coeff_monomial(E)
2
>>> p.nth(0, 1)
2
```

Since the following cannot be written as a product containing E as a factor, None is returned. (If the coefficient `2*x` is desired then the `coeff` method should be used.)

```
>>> (2*E*x + x).as_coefficient(E)
>>> (2*E*x + x).coeff(E)
2*x
```

```
>>> (E*(x + 1) + x).as_coefficient(E)
```

```
>>> (2*pi*I).as_coefficient(pi*I)
2
>>> (2*I).as_coefficient(pi*I)
```

as_coefficients_dict(**syms*)

Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0.

If symbols `syms` are provided, any multiplicative terms independent of them will be considered a coefficient and a regular dictionary of syms-dependent generators as keys and their corresponding coefficients as values will be returned.

Examples

```
>>> from sympy.abc import a, x, y
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
>>> (3*a*x).as_coefficients_dict(x)
{x: 3*a}
>>> (3*a*x).as_coefficients_dict(y)
{1: 3*a*x}
```

as_content_primitive(*radical=False*, *clear=True*)

This method should recursively remove a Rational from all arguments and return that (content) and the new self (primitive). The content should always be positive and `Mul(*foo.as_content_primitive()) == foo`. The primitive need not be in canonical form and should try to preserve the underlying structure if possible (i.e. expand_mul should not be applied to self).

Examples

```
>>> from sympy import sqrt
>>> from sympy.abc import x, y, z
```

```
>>> eq = 2 + 2*x + 2*y*(3 + 3*y)
```

The as_content_primitive function is recursive and retains structure:

```
>>> eq.as_content_primitive()
(2, x + 3*y*(y + 1) + 1)
```

Integer powers will have Rationals extracted from the base:

```
>>> ((2 + 6*x)**2).as_content_primitive()
(4, (3*x + 1)**2)
>>> ((2 + 6*x)**(2*y)).as_content_primitive()
(1, (2*(3*x + 1))**(2*y))
```

Terms may end up joining once their as_content_primitives are added:

```
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(11, x*(y + 1))
>>> ((3*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(9, x*(y + 1))
>>> ((3*(z*(1 + y)) + 2.0*x*(3 + 3*y))).as_content_primitive()
(1, 6.0*x*(y + 1) + 3*z*(y + 1))
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))**2).as_content_primitive()
(121, x**2*(y + 1)**2)
>>> ((x*(1 + y) + 0.4*x*(3 + 3*y))**2).as_content_primitive()
(1, 4.84*x**2*(y + 1)**2)
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

If clear=False (default is True) then content will not be removed from an Add if it can be distributed to leave one or more terms with integer coefficients.

```
>>> (x/2 + y).as_content_primitive()
(1/2, x + 2*y)
>>> (x/2 + y).as_content_primitive(clear=False)
(1, x/2 + y)
```

as_dummy()

Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True. When applied to a symbol a new symbol having only the same commutativity will be returned.

Notes

Any object that has structurally bound variables should have a property, *bound_symbols* that returns those symbols appearing in the object.

Examples

```
>>> from sympy import Integral, Symbol
>>> from sympy.abc import x
>>> r = Symbol('r', real=True)
>>> Integral(r, (r, x)).as_dummy()
Integral(_0, (_0, x))
>>> _.variables[0].is_real is None
True
>>> r.as_dummy()
_r
```

as_expr(**gens*)

Convert a polynomial to a SymPy expression.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y
```

```
>>> f = (x**2 + x*y).as_poly(x, y)
>>> f.as_expr()
x**2 + x*y
```

```
>>> sin(x).as_expr()
sin(x)
```

as_independent(**deps*, ***hint*) → tuple[Expr, Expr]

A mostly naive separation of a Mul or Add into arguments that are not are dependent on deps. To obtain as complete a separation of variables as possible, use a separation method first, e.g.:

- separatevars() to change Mul, Add and Pow (including exp) into Mul
- .expand(mul=True) to change Add or Mul into Add
- .expand(log=True) to change log expr into an Add

The only non-naive thing that is done here is to respect noncommutative ordering of variables and to always return (0, 0) for *self* of zero regardless of hints.

For nonzero *self*, the returned tuple (i, d) has the following interpretation:

- i will has no variable that appears in deps
- d will either have terms that contain variables that are in deps, or be equal to 0 (when self is an Add) or 1 (when self is a Mul)
- if self is an Add then self = i + d
- if self is a Mul then self = i*d
- otherwise (self, S.One) or (S.One, self) is returned.

To force the expression to be treated as an Add, use the hint as_Add=True

See also:

separatevars
expand_log
sympy.core.add.Add.as_two_terms
sympy.core.mul.Mul.as_two_terms
as_coeff_mul

Examples

– self is an Add

```
>>> from sympy import sin, cos, exp
>>> from sympy.abc import x, y, z
```

```
>>> (x + x*y).as_independent(x)
(0, x*y + x)
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> (2*x*sin(x) + y + x + z).as_independent(x)
(y + z, 2*x*sin(x) + x)
>>> (2*x*sin(x) + y + x + z).as_independent(x, y)
(z, 2*x*sin(x) + x + y)
```

– self is a Mul

```
>>> (x*sin(x)*cos(y)).as_independent(x)
(cos(y), x*sin(x))
```

non-commutative terms cannot always be separated out when self is a Mul

```
>>> from sympy import symbols
>>> n1, n2, n3 = symbols('n1 n2 n3', commutative=False)
>>> (n1 + n1*n2).as_independent(n2)
(n1, n1*n2)
>>> (n2*n1 + n1*n2).as_independent(n2)
(0, n1*n2 + n2*n1)
>>> (n1*n2*n3).as_independent(n1)
(1, n1*n2*n3)
>>> (n1*n2*n3).as_independent(n2)
(n1, n2*n3)
>>> ((x-n1)*(x-y)).as_independent(x)
(1, (x - y)*(x - n1))
```

– self is anything else:

```
>>> (sin(x)).as_independent(x)
(1, sin(x))
>>> (sin(x)).as_independent(y)
(sin(x), 1)
>>> exp(x+y).as_independent(x)
(1, exp(x + y))
```

– force self to be treated as an Add:

```
>>> (3*x).as_independent(x, as_Add=True)
(0, 3*x)
```

– force self to be treated as a Mul:

```
>>> (3+x).as_independent(x, as_Add=False)
(1, x + 3)
>>> (-3+x).as_independent(x, as_Add=False)
(1, x - 3)
```

Note how the below differs from the above in making the constant on the dep term positive.

```
>>> (y*(-3+x)).as_independent(x)
(y, x - 3)
```

– use .as_independent() for true independence testing instead
of .has(). The former considers only symbols in the free symbols while the latter considers all symbols

```
>>> from sympy import Integral
>>> I = Integral(x, (x, 1, 2))
>>> I.has(x)
True
>>> x in I.free_symbols
False
>>> I.as_independent(x) == (I, 1)
True
>>> (I + x).as_independent(x) == (I, x)
True
```

Note: when trying to get independent terms, a separation method might need to be used first. In this case, it is important to keep track of what you send to this routine so you know how to interpret the returned values

```
>>> from sympy import separatevars, log
>>> separatevars(exp(x+y)).as_independent(x)
(exp(y), exp(x))
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> separatevars(x + x*y).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).expand(mul=True).as_independent(y)
(x, x*y)
>>> a, b=symbols('a b', positive=True)
>>> (log(a*b).expand(log=True)).as_independent(b)
(log(a), log(b))
```

as_leading_term(**symbols, logx=None, cdir=0*)

Returns the leading (nonzero) term of the series expansion of self.

The _eval_as_leading_term routines are used to do this, and they must always return a non-zero value.

Examples

```
>>> from sympy.abc import x
>>> (1 + x + x**2).as_leading_term(x)
1
>>> (1/x**2 + x + x**2).as_leading_term(x)
x**(-2)
```

as_numer_denom()

Return the numerator and the denominator of an expression.

expression -> a/b -> a, b

This is just a stub that should be defined by an object's class methods to get anything else.

See also:

normal
    return a/b instead of (a, b)

as_ordered_factors(*order=None*)

Return list of ordered factors (if Mul) else [self].

as_ordered_terms(*order=None, data=False*)

Transform an expression to an ordered list of terms.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.abc import x
```

```
>>> (sin(x)**2*cos(x) + sin(x)**2 + 1).as_ordered_terms()
[sin(x)**2*cos(x), sin(x)**2, 1]
```

as_poly(**gens*, ***args*)

Converts `self` to a polynomial or returns `None`.

as_powers_dict()

Return self as a dictionary of factors with each factor being treated as a power. The keys are the bases of the factors and the values, the corresponding exponents. The resulting dictionary should be used with caution if the expression is a Mul and contains non- commutative factors since the order that they appeared will be lost in the dictionary.

See also:

as_ordered_factors
An alternative for noncommutative applications, returning an ordered list of factors.

args_cnc
Similar to as_ordered_factors, but guarantees separation of commutative and noncommutative factors.

as_real_imag(*deep=True*, ***hints*)

Performs complex expansion on 'self' and returns a tuple containing collected both real and imaginary parts. This method cannot be confused with re() and im() functions, which does not perform complex expansion at evaluation.

However it is possible to expand both re() and im() functions and get exactly the same results as with a single call to this function.

```
>>> from sympy import symbols, I
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> (x + y*I).as_real_imag()
(x, y)
```

```
>>> from sympy.abc import z, w
```

```
>>> (z + w*I).as_real_imag()
(re(z) - im(w), re(w) + im(z))
```

as_set()

Rewrites Boolean expression in terms of real sets.

Examples

```
>>> from sympy import Symbol, Eq, Or, And
>>> x = Symbol('x', real=True)
>>> Eq(x, 0).as_set()
{0}
>>> (x > 0).as_set()
Interval.open(0, oo)
>>> And(-2 < x, x < 2).as_set()
Interval.open(-2, 2)
>>> Or(x < -2, 2 < x).as_set()
Union(Interval.open(-oo, -2), Interval.open(2, oo))
```

`as_terms()`

Transform an expression to a list of terms.

`aseries`(*x=None*, *n=6*, *bound=0*, *hir=False*)

Asymptotic Series expansion of self. This is equivalent to `self.series(x, oo, n)`.

Parameters

self
[Expression] The expression whose series is to be expanded.

x
[Symbol] It is the variable of the expression to be calculated.

n
[Value] The value used to represent the order in terms of `x**n`, up to which the series is to be expanded.

hir
[Boolean] Set this parameter to be True to produce hierarchical series. It stops the recursion at an early level and may provide nicer and more useful results.

bound
[Value, Integer] Use the `bound` parameter to give limit on rewriting coefficients in its normalised form.

Returns

Expr
Asymptotic series expansion of the expression.

See also:

`Expr.aseries`
See the docstring of this function for complete details of this wrapper.

Notes

This algorithm is directly induced from the limit computational algorithm provided by Gruntz. It majorly uses the mrv and rewrite sub-routines. The overall idea of this algorithm is first to look for the most rapidly varying subexpression w of a given expression f and then expands f in a series in w. Then same thing is recursively done on the leading coefficient till we get constant coefficients.

If the most rapidly varying subexpression of a given expression f is f itself, the algorithm tries to find a normalised representation of the mrv set and rewrites f using this normalised representation.

If the expansion contains an order term, it will be either `O(x ** (-n))` or `O(w ** (-n))` where `w` belongs to the most rapidly varying expression of `self`.

References

[1], [2], [3]

Examples

```
>>> from sympy import sin, exp
>>> from sympy.abc import x
```

```
>>> e = sin(1/x + exp(-x)) - sin(1/x)
```

```
>>> e.aseries(x)
(1/(24*x**4) - 1/(2*x**2) + 1 + O(x**(-6), (x, oo)))*exp(-x)
```

```
>>> e.aseries(x, n=3, hir=True)
-exp(-2*x)*sin(1/x)/2 + exp(-x)*cos(1/x) + O(exp(-3*x), (x, oo))
```

```
>>> e = exp(exp(x)/(1 - 1/x))
```

```
>>> e.aseries(x)
exp(exp(x)/(1 - 1/x))
```

```
>>> e.aseries(x, bound=3)
exp(exp(x)/x**2)*exp(exp(x)/x)*exp(-exp(x) + exp(x)/(1 - 1/x) - exp(x)/x -␣
↪exp(x)/x**2)*exp(exp(x))
```

For rational expressions this method may return original expression without the Order term. >>> (1/x).aseries(x, n=8) 1/x

property assumptions0

Return object *type* assumptions.

For example:

Symbol('x', real=True) Symbol('x', integer=True)

are different objects. In other words, besides Python type (Symbol in this case), the initial assumptions are also forming their typeinfo.

Examples

```
>>> from sympy import Symbol
>>> from sympy.abc import x
>>> x.assumptions0
{'commutative': True}
>>> x = Symbol("x", positive=True)
>>> x.assumptions0
{'commutative': True, 'complex': True, 'extended_negative': False,
 'extended_nonnegative': True, 'extended_nonpositive': False,
 'extended_nonzero': True, 'extended_positive': True, 'extended_real':
 True, 'finite': True, 'hermitian': True, 'imaginary': False,
 'infinite': False, 'negative': False, 'nonnegative': True,
 'nonpositive': False, 'nonzero': True, 'positive': True, 'real':
 True, 'zero': False}
```

atoms(**types*)

Returns the atoms that form the current object.

By default, only objects that are truly atomic and cannot be divided into smaller pieces are returned: symbols, numbers, and number symbols like I and pi. It is possible to request atoms of any type, however, as demonstrated below.

Examples

```
>>> from sympy import I, pi, sin
>>> from sympy.abc import x, y
>>> (1 + x + 2*sin(y + I*pi)).atoms()
{1, 2, I, pi, x, y}
```

If one or more types are given, the results will contain only those types of atoms.

```
>>> from sympy import Number, NumberSymbol, Symbol
>>> (1 + x + 2*sin(y + I*pi)).atoms(Symbol)
{x, y}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number)
{1, 2}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol)
{1, 2, pi}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol, I)
{1, 2, I, pi}
```

Note that I (imaginary unit) and zoo (complex infinity) are special types of number symbols and are not part of the NumberSymbol class.

The type can be given implicitly, too:

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(x) # x is a Symbol
{x, y}
```

Be careful to check your assumptions when using the implicit option since `S(1).is_Integer = True` but `type(S(1))` is `One`, a special type of SymPy atom, while `type(S(2))` is type `Integer` and will find all integers in an expression:

```
>>> from sympy import S
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(1))
{1}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(2))
{1, 2}
```

Finally, arguments to atoms() can select more than atomic atoms: any SymPy type (loaded in core/__init__.py) can be listed as an argument and those types of "atoms" as found in scanning the arguments of the expression recursively:

```
>>> from sympy import Function, Mul
>>> from sympy.core.function import AppliedUndef
>>> f = Function('f')
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(Function)
{f(x), sin(y + I*pi)}
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(AppliedUndef)
{f(x)}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
{I*pi, 2*sin(y + I*pi)}
```

property binary_symbols

Return from the atoms of self those which are free symbols.

Not all free symbols are `Symbol`. Eg: IndexedBase('I')[0].free_symbols

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

cancel(**gens*, ***args*)

See the cancel function in sympy.polys

property canonical_variables

Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

Examples

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> Lambda(x, 2*x).canonical_variables
{x: _0}
```

classmethod class_key()

Nice order of classes.

coeff(*x*, *n=1*, *right=False*, *_first=True*)

Returns the coefficient from the term(s) containing `x**n`. If `n` is zero then all terms independent of `x` will be returned.

See also:

as_coefficient
separate the expression into a coefficient and factor

as_coeff_Add
separate the additive constant from an expression

as_coeff_Mul
separate the multiplicative constant from an expression

as_independent
separate x-dependent terms/factors from others

sympy.polys.polytools.Poly.coeff_monomial
efficiently find the single coefficient of a monomial in Poly

sympy.polys.polytools.Poly.nth
like coeff_monomial but powers of monomial terms are used

Examples

```
>>> from sympy import symbols
>>> from sympy.abc import x, y, z
```

You can select terms that have an explicit negative in front of them:

```
>>> (-x + 2*y).coeff(-1)
x
>>> (x - 2*y).coeff(-1)
2*y
```

You can select terms with no Rational coefficient:

```
>>> (x + 2*y).coeff(1)
x
>>> (3 + 2*x + 4*x**2).coeff(1)
0
```

You can select terms independent of x by making n=0; in this case expr.as_independent(x)[0] is returned (and 0 will be returned instead of None):

```
>>> (3 + 2*x + 4*x**2).coeff(x, 0)
3
>>> eq = ((x + 1)**3).expand() + 1
>>> eq
x**3 + 3*x**2 + 3*x + 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 2]
>>> eq -= 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 0]
```

You can select terms that have a numerical term in front of them:

```
>>> (-x - 2*y).coeff(2)
-y
>>> from sympy import sqrt
>>> (x + sqrt(2)*x).coeff(sqrt(2))
x
```

The matching is exact:

```
>>> (3 + 2*x + 4*x**2).coeff(x)
2
>>> (3 + 2*x + 4*x**2).coeff(x**2)
4
>>> (3 + 2*x + 4*x**2).coeff(x**3)
0
>>> (z*(x + y)**2).coeff((x + y)**2)
z
>>> (z*(x + y)**2).coeff(x + y)
0
```

In addition, no factoring is done, so 1 + z*(1 + y) is not obtained from the following:

```
>>> (x + z*(x + x*y)).coeff(x)
1
```

If such factoring is desired, factor_terms can be used first:

```
>>> from sympy import factor_terms
>>> factor_terms(x + z*(x + x*y)).coeff(x)
z*(y + 1) + 1
```

```
>>> n, m, o = symbols('n m o', commutative=False)
>>> n.coeff(n)
1
>>> (3*n).coeff(n)
3
>>> (n*m + m*n*m).coeff(n) # = (1 + m)*n*m
1 + m
>>> (n*m + m*n*m).coeff(n, right=True) # = (1 + m)*n*m
m
```

If there is more than one possible coefficient 0 is returned:

```
>>> (n*m + m*n).coeff(n)
0
```

If there is only one possible coefficient, it is returned:

```
>>> (n*m + x*m*n).coeff(m*n)
x
>>> (n*m + x*m*n).coeff(m*n, right=1)
1
```

collect(*syms*, *func=None*, *evaluate=True*, *exact=False*, *distribute_order_term=True*)
See the collect function in sympy.simplify

combsimp()
See the combsimp function in sympy.simplify

compare(*other*)
Return -1, 0, 1 if the object is smaller, equal, or greater than other.

Not in the mathematical sense. If the object is of a different type from the "other" then their classes are ordered according to the sorted_classes list.

Examples

```
>>> from sympy.abc import x, y
>>> x.compare(y)
-1
>>> x.compare(x)
0
>>> y.compare(x)
1
```

compute_leading_term(*x*, *logx=None*)
Deprecated function to compute the leading term of a series.

as_leading_term is only allowed for results of .series() This is a wrapper to compute a series first.

conjugate()
Returns the complex conjugate of 'self'.

copy()

could_extract_minus_sign()
Return True if self has -1 as a leading factor or has more literal negative signs than positive signs in a sum, otherwise False.

Examples

```
>>> from sympy.abc import x, y
>>> e = x - y
>>> {i.could_extract_minus_sign() for i in (e, -e)}
{False, True}
```

Though the `y - x` is considered like `-(x - y)`, since it is in a product without a leading factor of -1, the result is false below:

```
>>> (x*(y - x)).could_extract_minus_sign()
False
```

To put something in canonical form wrt to sign, use *signsimp*:

```
>>> from sympy import signsimp
>>> signsimp(x*(y - x))
-x*(x - y)
>>> _.could_extract_minus_sign()
True
```

count(*query*)

Count the number of matching subexpressions.

count_ops(*visual=None*)

Wrapper for count_ops that returns the operation count.

default_assumptions = {}

diff(**symbols*, ***assumptions*)

dir(*x*, *cdir*)

doit(***hints*)

Evaluate objects that are not evaluated by default like limits, integrals, sums and products. All objects of this kind will be evaluated recursively, unless some species were excluded via 'hints' or unless the 'deep' hint was set to 'False'.

```
>>> from sympy import Integral
>>> from sympy.abc import x
```

```
>>> 2*Integral(x, x)
2*Integral(x, x)
```

```
>>> (2*Integral(x, x)).doit()
x**2
```

```
>>> (2*Integral(x, x)).doit(deep=False)
2*Integral(x, x)
```

dummy_eq(*other*, *symbol=None*)

Compare two expressions and handle dummy symbols.

Examples

```
>>> from sympy import Dummy
>>> from sympy.abc import x, y
```

```
>>> u = Dummy('u')
```

```
>>> (u**2 + 1).dummy_eq(x**2 + 1)
True
>>> (u**2 + 1) == (x**2 + 1)
False
```

```
>>> (u**2 + y).dummy_eq(x**2 + y, x)
True
>>> (u**2 + y).dummy_eq(x**2 + y, y)
False
```

equals(*other*, *failing_expression=False*)

Return True if self == other, False if it does not, or None. If failing_expression is True then the expression which did not simplify to a 0 will be returned instead of None.

evalf(*n=15*, *subs=None*, *maxn=100*, *chop=False*, *strict=False*, *quad=None*, *verbose=False*)

Evaluate the given formula to an accuracy of *n* digits.

Parameters

subs
[dict, optional] Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

maxn
[int, optional] Allow a maximum temporary working precision of maxn digits.

chop
[bool or number, optional] Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When `True` the chop value defaults to standard precision.

Otherwise the chop value is used to determine the magnitude of "small" for purposes of chopping.

```
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
0.000100000000000000
>>> N(x, chop=1e-5)
0.000100000000000000
>>> N(x, chop=1e-4)
0
```

strict
[bool, optional] Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available maxprec.

quad
[str, optional] Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

verbose
[bool, optional] Print debug information.

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

expand(*deep=True*, *modulus=None*, *power_base=True*, *power_exp=True*, *mul=True*, *log=True*, *multinomial=True*, *basic=True*, ***hints*)

Expand an expression using hints.

See the docstring of the expand() function in sympy.core.function for more information.

property expr_free_symbols

Like `free_symbols`, but returns the free symbols only if they are contained in an expression node.

Examples

```
>>> from sympy.abc import x, y
>>> (x + y).expr_free_symbols
{x, y}
```

If the expression is contained in a non-expression object, do not return the free symbols. Compare:

```
>>> from sympy import Tuple
>>> t = Tuple(x + y)
>>> t.expr_free_symbols
set()
>>> t.free_symbols
{x, y}
```

extract_additively(*c*)

Return self - c if it's possible to subtract c from self and make all matching coefficients move towards zero, else return None.

See also:

extract_multiplicatively
coeff
as_coefficient

Examples

```
>>> from sympy.abc import x, y
>>> e = 2*x + 3
>>> e.extract_additively(x + 1)
x + 2
>>> e.extract_additively(3*x)
>>> e.extract_additively(4)
>>> (y*(x + 1)).extract_additively(x + 1)
>>> ((x + 1)*(x + 2*y + 1) + 3).extract_additively(x + 1)
(x + 1)*(x + 2*y) + 3
```

extract_branch_factor(*allow_half=False*)

Try to write self as `exp_polar(2*pi*I*n)*z` in a nice way. Return (z, n).

```
>>> from sympy import exp_polar, I, pi
>>> from sympy.abc import x, y
>>> exp_polar(I*pi).extract_branch_factor()
(exp_polar(I*pi), 0)
>>> exp_polar(2*I*pi).extract_branch_factor()
(1, 1)
>>> exp_polar(-pi*I).extract_branch_factor()
(exp_polar(I*pi), -1)
>>> exp_polar(3*pi*I + x).extract_branch_factor()
(exp_polar(x + I*pi), 1)
>>> (y*exp_polar(-5*pi*I)*exp_polar(3*pi*I + 2*pi*x)).extract_branch_factor()
(y*exp_polar(2*pi*x), -1)
>>> exp_polar(-I*pi/2).extract_branch_factor()
(exp_polar(-I*pi/2), 0)
```

If allow_half is True, also extract exp_polar(I*pi):

```
>>> exp_polar(I*pi).extract_branch_factor(allow_half=True)
(1, 1/2)
>>> exp_polar(2*I*pi).extract_branch_factor(allow_half=True)
(1, 1)
>>> exp_polar(3*I*pi).extract_branch_factor(allow_half=True)
(1, 3/2)
>>> exp_polar(-I*pi).extract_branch_factor(allow_half=True)
(1, -1/2)
```

extract_multiplicatively(*c*)

Return None if it's not possible to make self in the form c * something in a nice way, i.e. preserving the properties of arguments of self.

Examples

```
>>> from sympy import symbols, Rational
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> ((x*y)**3).extract_multiplicatively(x**2 * y)
x*y**2
```

```
>>> ((x*y)**3).extract_multiplicatively(x**4 * y)
```

```
>>> (2*x).extract_multiplicatively(2)
x
```

```
>>> (2*x).extract_multiplicatively(3)
```

```
>>> (Rational(1, 2)*x).extract_multiplicatively(3)
x/6
```

factor(**gens*, ***args*)

See the factor() function in sympy.polys.polytools

find(*query*, *group=False*)

Find all subexpressions matching a query.

property formula

Return a Formula with only terms=[self].

fourier_series(*limits=None*)

Compute fourier sine/cosine series of self.

See the docstring of the `fourier_series()` in sympy.series.fourier for more information.

fps(*x=None*, *x0=0*, *dir=1*, *hyper=True*, *order=4*, *rational=True*, *full=False*)

Compute formal power power series of self.

See the docstring of the `fps()` function in sympy.series.formal for more information.

property free_symbols

Return from the atoms of self those which are free symbols.

Not all free symbols are `Symbol`. Eg: IndexedBase('I')[0].free_symbols

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

classmethod fromiter(*args*, ***assumptions*)

Create a new object from an iterable.

This is a convenience function that allows one to create objects from any iterable, without having to convert to a list or tuple first.

Examples

```
>>> from sympy import Tuple
>>> Tuple.fromiter(i for i in range(5))
(0, 1, 2, 3, 4)
```

property func

The top-level function in an expression.

The following should hold for all objects:

```
>> x == x.func(*x.args)
```

Examples

```
>>> from sympy.abc import x
>>> a = 2*x
>>> a.func
<class 'sympy.core.mul.Mul'>
>>> a.args
(2, x)
>>> a.func(*a.args)
2*x
>>> a == a.func(*a.args)
True
```

gammasimp()

See the gammasimp function in sympy.simplify

getO()

Returns the additive O(..) symbol if there is one, else None.

getn()

Returns the order of the expression.

Examples

```
>>> from sympy import O
>>> from sympy.abc import x
>>> (1 + x + O(x**2)).getn()
2
>>> (1 + x).getn()
```

has(**patterns*)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y, z
>>> (x**2 + sin(x*y)).has(z)
False
>>> (x**2 + sin(x*y)).has(x, y, z)
True
>>> x.has(x)
True
```

Note has is a structural algorithm with no knowledge of mathematics. Consider the following half-open interval:

```
>>> from sympy import Interval
>>> i = Interval.Lopen(0, 5); i
Interval.Lopen(0, 5)
>>> i.args
(0, 5, True, False)
>>> i.has(4)  # there is no "4" in the arguments
False
>>> i.has(0)  # there *is* a "0" in the arguments
True
```

Instead, use contains to determine whether a number is in the interval or not:

```
>>> i.contains(4)
True
>>> i.contains(0)
False
```

Note that `expr.has(*patterns)` is exactly equivalent to `any(expr.has(p) for p in patterns)`. In particular, `False` is returned when the list of patterns is empty.

```
>>> x.has()
False
```

has_free(**patterns*)

Return True if self has object(s) x as a free expression else False.

Examples

```
>>> from sympy import Integral, Function
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> g = Function('g')
>>> expr = Integral(f(x), (f(x), 1, g(y)))
>>> expr.free_symbols
{y}
>>> expr.has_free(g(y))
True
>>> expr.has_free(*(x, f(x)))
False
```

This works for subexpressions and types, too:

```
>>> expr.has_free(g)
True
>>> (x + y + 1).has_free(y + 1)
True
```

has_xfree(*s: set[Basic]*)

Return True if self has any of the patterns in s as a free argument, else False. This is like *Basic.has_free* but this will only report exact argument matches.

Examples

```
>>> from sympy import Function
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> f(x).has_xfree({f})
False
>>> f(x).has_xfree({f(x)})
True
>>> f(x + 1).has_xfree({x})
True
>>> f(x + 1).has_xfree({x + 1})
True
>>> f(x + y + 1).has_xfree({x + 1})
False
```

integrate(**args*, ***kwargs*)

See the integrate function in sympy.integrals

invert(*g*, **gens*, ***args*)

Return the multiplicative inverse of `self` mod `g` where `self` (and `g`) may be symbolic expressions).

See also:

`sympy.core.numbers.mod_inverse`, `sympy.polys.polytools.invert`

is_Add = False

is_AlgebraicNumber = False

is_Atom = True

is_Boolean = False

is_Derivative = False

is_Dummy = False

is_Equality = False

is_Float = False

is_Function = False

`is_Indexed = False`

`is_Integer = False`

`is_MatAdd = False`

`is_MatMul = False`

`is_Matrix = False`

`is_Mul = False`

`is_Not = False`

`is_Number = False`

`is_NumberSymbol = False`

`is_Order = False`

`is_Piecewise = False`

`is_Point = False`

`is_Poly = False`

`is_Pow = False`

`is_Rational = False`

`is_Relational = False`

`is_Symbol = True`

`is_Vector = False`

`is_Wild = False`

`property is_algebraic`

`is_algebraic_expr`(**syms*)

This tests whether a given expression is algebraic or not, in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are "algebraic expressions" with symbolic exponents. This is a simple extension to the is_rational_function, including rational exponentiation.

See also:

is_rational_function

References

[1]

Examples

```
>>> from sympy import Symbol, sqrt
>>> x = Symbol('x', real=True)
>>> sqrt(1 + x).is_rational_function()
False
>>> sqrt(1 + x).is_algebraic_expr()
True
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be an algebraic expression to become one.

```
>>> from sympy import exp, factor
>>> a = sqrt(exp(x)**2 + 2*exp(x) + 1)/(exp(x) + 1)
>>> a.is_algebraic_expr(x)
False
>>> factor(a).is_algebraic_expr()
True
```

property is_antihermitian

property is_commutative

is_comparable = False

property is_complex

property is_composite

is_constant(**wrt*, ***flags*)

Return True if self is constant, False if not, or None if the constancy could not be determined conclusively.

Examples

```
>>> from sympy import cos, sin, Sum, S, pi
>>> from sympy.abc import a, n, x, y
>>> x.is_constant()
False
>>> S(2).is_constant()
True
>>> Sum(x, (x, 1, 10)).is_constant()
True
>>> Sum(x, (x, 1, n)).is_constant()
False
>>> Sum(x, (x, 1, n)).is_constant(y)
True
>>> Sum(x, (x, 1, n)).is_constant(n)
False
```

(continues on next page)

(continued from previous page)

```
>>> Sum(x, (x, 1, n)).is_constant(x)
True
>>> eq = a*cos(x)**2 + a*sin(x)**2 - a
>>> eq.is_constant()
True
>>> eq.subs({x: pi, a: 2}) == eq.subs({x: pi, a: 3}) == 0
True
```

```
>>> (0**x).is_constant()
False
>>> x.is_constant()
False
>>> (x**x).is_constant()
False
>>> one = cos(x)**2 + sin(x)**2
>>> one.is_constant()
True
>>> ((one - 1)**(x + 1)).is_constant() in (True, False) # could be 0 or 1
True
```

property is_even

property is_extended_negative

property is_extended_nonnegative

property is_extended_nonpositive

property is_extended_nonzero

property is_extended_positive

property is_extended_real

property is_finite

property is_hermitian

is_hypergeometric(*k*)

property is_imaginary

property is_infinite

property is_integer

property is_irrational

is_meromorphic(*x*, *a*)

This tests whether an expression is meromorphic as a function of the given symbol `x` at the point `a`.

This method is intended as a quick test that will return None if no decision can be made without simplification or more detailed analysis.

Examples

```
>>> from sympy import zoo, log, sin, sqrt
>>> from sympy.abc import x
```

```
>>> f = 1/x**2 + 1 - 2*x**3
>>> f.is_meromorphic(x, 0)
True
>>> f.is_meromorphic(x, 1)
True
>>> f.is_meromorphic(x, zoo)
True
```

```
>>> g = x**log(3)
>>> g.is_meromorphic(x, 0)
False
>>> g.is_meromorphic(x, 1)
True
>>> g.is_meromorphic(x, zoo)
False
```

```
>>> h = sin(1/x)*x**2
>>> h.is_meromorphic(x, 0)
False
>>> h.is_meromorphic(x, 1)
True
>>> h.is_meromorphic(x, zoo)
True
```

Multivalued functions are considered meromorphic when their branches are meromorphic. Thus most functions are meromorphic everywhere except at essential singularities and branch points. In particular, they will be meromorphic also on branch cuts except at their endpoints.

```
>>> log(x).is_meromorphic(x, -1)
True
>>> log(x).is_meromorphic(x, 0)
False
>>> sqrt(x).is_meromorphic(x, -1)
True
>>> sqrt(x).is_meromorphic(x, 0)
False
```

property is_negative

property is_noninteger

property is_nonnegative

property is_nonpositive

property is_nonzero

is_number = False

property is_odd

property is_polar

is_polynomial(**syms*)

Return True if self is a polynomial in syms and False otherwise.

This checks if self is an exact polynomial in syms. This function returns False for expressions that are "polynomials" with symbolic exponents. Thus, you should be able to apply polynomial algorithms to expressions for which this returns True, and Poly(expr, *syms) should work if and only if expr.is_polynomial(*syms) returns True. The polynomial does not have to be in expanded form. If no symbols are given, all free symbols in the expression will be used.

This is not part of the assumptions system. You cannot do Symbol('z', polynomial=True).

Examples

```
>>> from sympy import Symbol, Function
>>> x = Symbol('x')
>>> ((x**2 + 1)**4).is_polynomial(x)
True
>>> ((x**2 + 1)**4).is_polynomial()
True
>>> (2**x + 1).is_polynomial(x)
False
>>> (2**x + 1).is_polynomial(2**x)
True
>>> f = Function('f')
>>> (f(x) + 1).is_polynomial(x)
False
>>> (f(x) + 1).is_polynomial(f(x))
True
>>> (1/f(x) + 1).is_polynomial(f(x))
False
```

```
>>> n = Symbol('n', nonnegative=True, integer=True)
>>> (x**n + 1).is_polynomial(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a polynomial to become one.

```
>>> from sympy import sqrt, factor, cancel
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)
>>> a.is_polynomial(y)
False
>>> factor(a)
y + 1
>>> factor(a).is_polynomial(y)
True
```

```
>>> b = (y**2 + 2*y + 1)/(y + 1)
>>> b.is_polynomial(y)
False
>>> cancel(b)
y + 1
>>> cancel(b).is_polynomial(y)
True
```

See also .is_rational_function()

property is_positive

property is_prime

property is_rational

is_rational_function(**syms*)

Test whether function is a ratio of two polynomials in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are "rational functions" with symbolic exponents. Thus, you should be able to call .as_numer_denom() and apply polynomial algorithms to the result for expressions for which this returns True.

This is not part of the assumptions system. You cannot do Symbol('z', rational_function=True).

Examples

```
>>> from sympy import Symbol, sin
>>> from sympy.abc import x, y
```

```
>>> (x/y).is_rational_function()
True
```

```
>>> (x**2).is_rational_function()
True
```

```
>>> (x/sin(y)).is_rational_function(y)
False
```

```
>>> n = Symbol('n', integer=True)
>>> (x**n + 1).is_rational_function(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a rational function to become one.

```
>>> from sympy import sqrt, factor
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)/y
>>> a.is_rational_function(y)
False
```

(continues on next page)

(continued from previous page)

```
>>> factor(a)
(y + 1)/y
>>> factor(a).is_rational_function(y)
True
```

See also is_algebraic_expr().

property is_real

is_scalar = True

is_symbol = True

property is_transcendental

property is_zero

property kind

Default kind for all SymPy object. If the kind is not defined for the object, or if the object cannot infer the kind from its arguments, this will be returned.

Examples

```
>>> from sympy import Expr
>>> Expr().kind
UndefinedKind
```

leadterm(*x*, *logx=None*, *cdir=0*)

Returns the leading term a*x**b as a tuple (a, b).

Examples

```
>>> from sympy.abc import x
>>> (1+x+x**2).leadterm(x)
(1, 0)
>>> (1/x**2+x+x**2).leadterm(x)
(1, -2)
```

limit(*x*, *xlim*, *dir='+'*)

Compute limit x->xlim.

lseries(*x=None*, *x0=0*, *dir='+'*, *logx=None*, *cdir=0*)

Wrapper for series yielding an iterator of the terms of the series.

Note: an infinite series will yield an infinite iterator. The following, for exaxmple, will never terminate. It will just keep printing terms of the sin(x) series:

```
for term in sin(x).lseries(x):
    print term
```

The advantage of lseries() over nseries() is that many times you are just interested in the next term in the series (i.e. the first term for example), but you do not know how many you should ask for in nseries() using the "n" parameter.

See also nseries().

match(*pattern*, *old=False*)

Pattern matching.

Wild symbols match all.

Return None when expression (self) does not match with pattern. Otherwise return a dictionary such that:

```
pattern.xreplace(self.match(pattern)) == self
```

Examples

```
>>> from sympy import Wild, Sum
>>> from sympy.abc import x, y
>>> p = Wild("p")
>>> q = Wild("q")
>>> r = Wild("r")
>>> e = (x+y)**(x+y)
>>> e.match(p**p)
{p_: x + y}
>>> e.match(p**q)
{p_: x + y, q_: x + y}
>>> e = (2*x)**2
>>> e.match(p*q**r)
{p_: 4, q_: x, r_: 2}
>>> (p*q**r).xreplace(e.match(p*q**r))
4*x**2
```

Structurally bound symbols are ignored during matching:

```
>>> Sum(x, (x, 1, 2)).match(Sum(y, (y, 1, p)))
{p_: 2}
```

But they can be identified if desired:

```
>>> Sum(x, (x, 1, 2)).match(Sum(q, (q, 1, p)))
{p_: 2, q_: x}
```

The old flag will give the old-style pattern matching where expressions and patterns are essentially solved to give the match. Both of the following give None unless old=True:

```
>>> (x - 2).match(p - x, old=True)
{p_: 2*x - 2}
>>> (2/x).match(p*x, old=True)
{p_: 2/x**2}
```

matches(*expr*, *repl_dict=None*, *old=False*)

Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.

Examples

```
>>> from sympy import symbols, Wild, Basic
>>> a, b, c = symbols('a b c')
>>> x = Wild('x')
>>> Basic(a + x, x).matches(Basic(a + b, c)) is None
True
>>> Basic(a + x, x).matches(Basic(a + b + c, b + c))
{x_: b + c}
```

n(*n=15*, *subs=None*, *maxn=100*, *chop=False*, *strict=False*, *quad=None*, *verbose=False*)

Evaluate the given formula to an accuracy of *n* digits.

Parameters

subs
[dict, optional] Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

maxn
[int, optional] Allow a maximum temporary working precision of maxn digits.

chop
[bool or number, optional] Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When `True` the chop value defaults to standard precision.

Otherwise the chop value is used to determine the magnitude of “small” for purposes of chopping.

```
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
0.000100000000000000
>>> N(x, chop=1e-5)
0.000100000000000000
>>> N(x, chop=1e-4)
0
```

strict
[bool, optional] Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available maxprec.

quad
[str, optional] Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

verbose
[bool, optional] Print debug information.

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

name: str

normal()

Return the expression as a fraction.

expression -> a/b

See also:

as_numer_denom
return `(a, b)` instead of `a/b`

nseries(*x=None*, *x0=0*, *n=6*, *dir='+'*, *logx=None*, *cdir=0*)

Wrapper to _eval_nseries if assumptions allow, else to series.

If x is given, x0 is 0, dir='+', and self has x, then _eval_nseries is called. This calculates "n" terms in the innermost expressions and then builds up the final series just by "cross-multiplying" everything out.

The optional `logx` parameter can be used to replace any log(x) in the returned series with a symbolic value to avoid evaluating log(x) at 0. A symbol to use in place of log(x) should be provided.

Advantage – it's fast, because we do not have to determine how many terms we need to calculate in advance.

Disadvantage – you may end up with less terms than you may have expected, but the O(x**n) term appended will always be correct and so the result, though perhaps shorter, will also be correct.

If any of those assumptions is not met, this is treated like a wrapper to series which will try harder to return the correct number of terms.

See also lseries().

Examples

```
>>> from sympy import sin, log, Symbol
>>> from sympy.abc import x, y
>>> sin(x).nseries(x, 0, 6)
x - x**3/6 + x**5/120 + O(x**6)
>>> log(x+1).nseries(x, 0, 5)
x - x**2/2 + x**3/3 - x**4/4 + O(x**5)
```

Handling of the `logx` parameter — in the following example the expansion fails since `sin` does not have an asymptotic expansion at -oo (the limit of log(x) as x approaches 0):

```
>>> e = sin(log(x))
>>> e.nseries(x, 0, 6)
Traceback (most recent call last):
...
PoleError: ...
...
>>> logx = Symbol('logx')
>>> e.nseries(x, 0, 6, logx=logx)
sin(logx)
```

In the following example, the expansion works but only returns self unless the `logx` parameter is used:

```
>>> e = x**y
>>> e.nseries(x, 0, 2)
x**y
>>> e.nseries(x, 0, 2, logx=logx)
exp(logx*y)
```

nsimplify(*constants=()*, *tolerance=None*, *full=False*)

See the nsimplify function in sympy.simplify

powsimp(**args*, ***kwargs*)

See the powsimp function in sympy.simplify

primitive()

Return the positive Rational that can be extracted non-recursively from every term of self (i.e., self is treated like an Add). This is like the as_coeff_Mul() method but primitive always extracts a positive Rational (never a negative or a Float).

Examples

```
>>> from sympy.abc import x
>>> (3*(x + 1)**2).primitive()
(3, (x + 1)**2)
>>> a = (6*x + 2); a.primitive()
(2, 3*x + 1)
>>> b = (x/2 + 3); b.primitive()
(1/2, x + 6)
>>> (a*b).primitive() == (1, a*b)
True
```

radsimp(***kwargs*)

See the radsimp function in sympy.simplify

ratsimp()

See the ratsimp function in sympy.simplify

rcall(**args*)

Apply on the argument recursively through the expression tree.

This method is used to simulate a common abuse of notation for operators. For instance, in SymPy the following will not work:

(x+Lambda(y, 2*y))(z) == x+2*z,

however, you can use:

```
>>> from sympy import Lambda
>>> from sympy.abc import x, y, z
>>> (x + Lambda(y, 2*y)).rcall(z)
x + 2*z
```

refine(*assumption=True*)

See the refine function in sympy.assumptions

removeO()

Removes the additive O(..) symbol if there is one

replace(*query*, *value*, *map=False*, *simultaneous=True*, *exact=None*)

Replace matching subexpressions of `self` with `value`.

If `map = True` then also return the mapping {old: new} where `old` was a sub-expression found with query and `new` is the replacement value for it. If the expression itself does not match the query, then the returned value will be `self.xreplace(map)` otherwise it should be `self.subs(ordered(map.items()))`.

Traverses an expression tree and performs replacement of matching subexpressions from the bottom to the top of the tree. The default approach is to do the replacement in a simultaneous fashion so changes made are targeted only once. If this is not desired or causes problems, `simultaneous` can be set to False.

In addition, if an expression containing more than one Wild symbol is being used to match subexpressions and the `exact` flag is None it will be set to True so the match will only succeed if all non-zero values are received for each Wild that appears in the match pattern. Setting this to False accepts a match of 0; while setting it True accepts all matches that have a 0 in them. See example below for cautions.

The list of possible combinations of queries and replacement values is listed below:

See also:

subs
substitution of subexpressions as defined by the objects themselves.

xreplace
exact node replacement in expr tree; also capable of using matching rules

Examples

Initial setup

```
>>> from sympy import log, sin, cos, tan, Wild, Mul, Add
>>> from sympy.abc import x, y
>>> f = log(sin(x)) + tan(sin(x**2))
```

1.1. type -> type
obj.replace(type, newtype)

When object of type `type` is found, replace it with the result of passing its argument(s) to `newtype`.

```
>>> f.replace(sin, cos)
log(cos(x)) + tan(cos(x**2))
>>> sin(x).replace(sin, cos, map=True)
```

(continues on next page)

(continued from previous page)

```
(cos(x), {sin(x): cos(x)})
>>> (x*y).replace(Mul, Add)
x + y
```

1.2. type -> func
obj.replace(type, func)

When object of type `type` is found, apply `func` to its argument(s). `func` must be written to handle the number of arguments of `type`.

```
>>> f.replace(sin, lambda arg: sin(2*arg))
log(sin(2*x)) + tan(sin(2*x**2))
>>> (x*y).replace(Mul, lambda *args: sin(2*Mul(*args)))
sin(2*x*y)
```

2.1. pattern -> expr
obj.replace(pattern(wild), expr(wild))

Replace subexpressions matching `pattern` with the expression written in terms of the Wild symbols in `pattern`.

```
>>> a, b = map(Wild, 'ab')
>>> f.replace(sin(a), tan(a))
log(tan(x)) + tan(tan(x**2))
>>> f.replace(sin(a), tan(a/2))
log(tan(x/2)) + tan(tan(x**2/2))
>>> f.replace(sin(a), a)
log(x) + tan(x**2)
>>> (x*y).replace(a*x, a)
y
```

Matching is exact by default when more than one Wild symbol is used: matching fails unless the match gives non-zero values for all Wild symbols:

```
>>> (2*x + y).replace(a*x + b, b - a)
y - 2
>>> (2*x).replace(a*x + b, b - a)
2*x
```

When set to False, the results may be non-intuitive:

```
>>> (2*x).replace(a*x + b, b - a, exact=False)
2/x
```

2.2. pattern -> func
obj.replace(pattern(wild), lambda wild: expr(wild))

All behavior is the same as in 2.1 but now a function in terms of pattern variables is used rather than an expression:

```
>>> f.replace(sin(a), lambda a: sin(2*a))
log(sin(2*x)) + tan(sin(2*x**2))
```

3.1. func -> func
obj.replace(filter, func)

Replace subexpression e with func(e) if filter(e) is True.

```
>>> g = 2*sin(x**3)
>>> g.replace(lambda expr: expr.is_Number, lambda expr: expr**2)
4*sin(x**9)
```

The expression itself is also targeted by the query but is done in such a fashion that changes are not made twice.

```
>>> e = x*(x*y + 1)
>>> e.replace(lambda x: x.is_Mul, lambda x: 2*x)
2*x*(2*x*y + 1)
```

When matching a single symbol, *exact* will default to True, but this may or may not be the behavior that is desired:

Here, we want *exact=False*:

```
>>> from sympy import Function
>>> f = Function('f')
>>> e = f(1) + f(0)
>>> q = f(a), lambda a: f(a + 1)
>>> e.replace(*q, exact=False)
f(1) + f(2)
>>> e.replace(*q, exact=True)
f(0) + f(2)
```

But here, the nature of matching makes selecting the right setting tricky:

```
>>> e = x**(1 + y)
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**-a, exact=False)
x
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**-a, exact=True)
x**(-x - y + 1)
>>> (x**y).replace(x**(1 + a), lambda a: x**-a, exact=False)
x
>>> (x**y).replace(x**(1 + a), lambda a: x**-a, exact=True)
x**(1 - y)
```

It is probably better to use a different form of the query that describes the target expression more precisely:

```
>>> (1 + x**(1 + y)).replace(
... lambda x: x.is_Pow and x.exp.is_Add and x.exp.args[0] == 1,
... lambda x: x.base**(1 - (x.exp - 1)))
...
x**(1 - y) + 1
```

rewrite(**args*, *deep=True*, ***hints*)

Rewrite *self* using a defined rule.

Rewriting transforms an expression to another, which is mathematically equivalent but structurally different. For example you can rewrite trigonometric functions as complex exponentials or combinatorial functions as gamma function.

This method takes a *pattern* and a *rule* as positional arguments. *pattern* is optional parameter which defines the types of expressions that will be transformed. If it is not passed, all possible expressions will be rewritten. *rule* defines how the expression will be rewritten.

Parameters

args
[Expr] A *rule*, or *pattern* and *rule*. - *pattern* is a type or an iterable of types. - *rule* can be any object.

deep
[bool, optional] If `True`, subexpressions are recursively transformed. Default is `True`.

Examples

If *pattern* is unspecified, all possible expressions are transformed.

```
>>> from sympy import cos, sin, exp, I
>>> from sympy.abc import x
>>> expr = cos(x) + I*sin(x)
>>> expr.rewrite(exp)
exp(I*x)
```

Pattern can be a type or an iterable of types.

```
>>> expr.rewrite(sin, exp)
exp(I*x)/2 + cos(x) - exp(-I*x)/2
>>> expr.rewrite([cos,], exp)
exp(I*x)/2 + I*sin(x) + exp(-I*x)/2
>>> expr.rewrite([cos, sin], exp)
exp(I*x)
```

Rewriting behavior can be implemented by defining `_eval_rewrite()` method.

```
>>> from sympy import Expr, sqrt, pi
>>> class MySin(Expr):
...     def _eval_rewrite(self, rule, args, **hints):
...         x, = args
...         if rule == cos:
...             return cos(pi/2 - x, evaluate=False)
...         if rule == sqrt:
...             return sqrt(1 - cos(x)**2)
>>> MySin(MySin(x)).rewrite(cos)
cos(-cos(-x + pi/2) + pi/2)
>>> MySin(x).rewrite(sqrt)
sqrt(1 - cos(x)**2)
```

Defining `_eval_rewrite_as_[...]()` method is supported for backwards compatibility reason. This may be removed in the future and using it is discouraged.

```
>>> class MySin(Expr):
...     def _eval_rewrite_as_cos(self, *args, **hints):
...         x, = args
...         return cos(pi/2 - x, evaluate=False)
>>> MySin(x).rewrite(cos)
cos(-x + pi/2)
```

round(*n=None*)

Return x rounded to the given decimal place.

If a complex number would results, apply round to the real and imaginary components of the number.

Notes

The Python `round` function uses the SymPy `round` method so it will always return a SymPy number (not a Python float or int):

```
>>> isinstance(round(S(123), -2), Number)
True
```

Examples

```
>>> from sympy import pi, E, I, S, Number
>>> pi.round()
3
>>> pi.round(2)
3.14
>>> (2*pi + E*I).round()
6 + 3*I
```

The round method has a chopping effect:

```
>>> (2*pi + I/10).round()
6
>>> (pi/10 + 2*I).round()
2*I
>>> (pi/10 + E*I).round(2)
0.31 + 2.72*I
```

separate(*deep=False*, *force=False*)

See the separate function in sympy.simplify

series(*x=None*, *x0=0*, *n=6*, *dir='+'*, *logx=None*, *cdir=0*)

Series expansion of "self" around `x = x0` yielding either terms of the series one by one (the lazy series given when n=None), else all the terms at once when n != None.

Returns the series expansion of "self" around the point `x = x0` with respect to `x` up to `O((x - x0)**n, x, x0)` (default n is 6).

If `x=None` and `self` is univariate, the univariate symbol will be supplied, otherwise an error will be raised.

Parameters

expr
[Expression] The expression whose series is to be expanded.

x
[Symbol] It is the variable of the expression to be calculated.

x0
[Value] The value around which `x` is calculated. Can be any value from `-oo` to `oo`.

n
[Value] The value used to represent the order in terms of `x**n`, up to which the series is to be expanded.

dir
[String, optional] The series-expansion can be bi-directional. If `dir="+"`, then (x->x0+). If `dir="-", then (x->x0-). For infinite ``x0` (`oo` or `-oo`), the `dir` argument is determined from the direction of the infinity (i.e., `dir="-"` for `oo`).

logx
[optional] It is used to replace any log(x) in the returned series with a symbolic value rather than evaluating the actual value.

cdir
[optional] It stands for complex direction, and indicates the direction from which the expansion needs to be evaluated.

Returns

Expr
[Expression] Series expansion of the expression about x0

Raises

TypeError
If “n” and “x0” are infinity objects

PoleError
If “x0” is an infinity object

Examples

```
>>> from sympy import cos, exp, tan
>>> from sympy.abc import x, y
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
>>> cos(x).series(n=4)
1 - x**2/2 + O(x**4)
>>> cos(x).series(x, x0=1, n=2)
cos(1) - (x - 1)*sin(1) + O((x - 1)**2, (x, 1))
>>> e = cos(x + exp(y))
>>> e.series(y, n=2)
cos(x + 1) - y*sin(x + 1) + O(y**2)
>>> e.series(x, n=2)
cos(exp(y)) - x*sin(exp(y)) + O(x**2)
```

If `n=None` then a generator of the series terms will be returned.

```
>>> term=cos(x).series(n=None)
>>> [next(term) for i in range(2)]
[1, -x**2/2]
```

For `dir=+` (default) the series is calculated from the right and for `dir=-` the series from the left. For smooth functions this flag will not alter the results.

```
>>> abs(x).series(dir="+")
x
>>> abs(x).series(dir="-")
-x
>>> f = tan(x)
>>> f.series(x, 2, 6, "+")
tan(2) + (1 + tan(2)**2)*(x - 2) + (x - 2)**2*(tan(2)**3 + tan(2)) +
(x - 2)**3*(1/3 + 4*tan(2)**2/3 + tan(2)**4) + (x - 2)**4*(tan(2)**5 +
5*tan(2)**3/3 + 2*tan(2)/3) + (x - 2)**5*(2/15 + 17*tan(2)**2/15 +
2*tan(2)**4 + tan(2)**6) + O((x - 2)**6, (x, 2))
```

```
>>> f.series(x, 2, 3, "-")
tan(2) + (2 - x)*(-tan(2)**2 - 1) + (2 - x)**2*(tan(2)**3 + tan(2))
+ O((x - 2)**3, (x, 2))
```

For rational expressions this method may return original expression without the Order term. >>> (1/x).series(x, n=8) 1/x

simplify(***kwargs*)

See the simplify function in sympy.simplify

sort_key(*order=None*)

Return a sort key.

Examples

```
>>> from sympy import S, I
```

```
>>> sorted([S(1)/2, I, -I], key=lambda x: x.sort_key())
[1/2, -I, I]
```

```
>>> S("[x, 1/x, 1/x**2, x**2, x**(1/2), x**(1/4), x**(3/2)]")
[x, 1/x, x**(-2), x**2, sqrt(x), x**(1/4), x**(3/2)]
>>> sorted(_, key=lambda x: x.sort_key())
[x**(-2), 1/x, x**(1/4), sqrt(x), x, x**(3/2), x**2]
```

subs(**args*, ***kwargs*)

Substitutes old for new in an expression after sympifying args.

***args* is either:**

- two arguments, e.g. foo.subs(old, new)
- **one iterable argument, e.g. foo.subs(iterable). The iterable may be**

  o an iterable container with (old, new) pairs. In this case the
  replacements are processed in the order given with successive patterns possibly affecting replacements already made.

  o a dict or set whose key/value items correspond to old/new pairs.
  In this case the old/new pairs will be sorted by op count and in case of a tie, by number of args and the default_sort_key. The resulting sorted list is then processed as an iterable container (see previous).

If the keyword `simultaneous` is True, the subexpressions will not be evaluated until all the substitutions have been made.

See also:

replace
: replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

xreplace
: exact node replacement in expr tree; also capable of using matching rules

`sympy.core.evalf.EvalfMixin.evalf`
: calculates the given formula to a desired level of precision

Examples

```
>>> from sympy import pi, exp, limit, oo
>>> from sympy.abc import x, y
>>> (1 + x*y).subs(x, pi)
pi*y + 1
>>> (1 + x*y).subs({x:pi, y:2})
1 + 2*pi
>>> (1 + x*y).subs([(x, pi), (y, 2)])
1 + 2*pi
>>> reps = [(y, x**2), (x, 2)]
>>> (x + y).subs(reps)
6
>>> (x + y).subs(reversed(reps))
x**2 + 2
```

```
>>> (x**2 + x**4).subs(x**2, y)
y**2 + y
```

To replace only the x**2 but not the x**4, use xreplace:

```
>>> (x**2 + x**4).xreplace({x**2: y})
x**4 + y
```

To delay evaluation until all substitutions have been made, set the keyword `simultaneous` to True:

```
>>> (x/y).subs([(x, 0), (y, 0)])
0
>>> (x/y).subs([(x, 0), (y, 0)], simultaneous=True)
nan
```

This has the added feature of not allowing subsequent substitutions to affect those already made:

```
>>> ((x + y)/y).subs({x + y: y, y: x + y})
1
>>> ((x + y)/y).subs({x + y: y, y: x + y}, simultaneous=True)
y/(x + y)
```

In order to obtain a canonical result, unordered iterables are sorted by count_op length, number of arguments and by the default_sort_key to break any ties. All other iterables are left unsorted.

```
>>> from sympy import sqrt, sin, cos
>>> from sympy.abc import a, b, c, d, e
```

```
>>> A = (sqrt(sin(2*x)), a)
>>> B = (sin(2*x), b)
>>> C = (cos(2*x), c)
>>> D = (x, d)
>>> E = (exp(x), e)
```

```
>>> expr = sqrt(sin(2*x))*sin(exp(x)*x)*cos(2*x) + sin(2*x)
```

```
>>> expr.subs(dict([A, B, C, D, E]))
a*c*sin(d*e) + b
```

The resulting expression represents a literal replacement of the old arguments with the new arguments. This may not reflect the limiting behavior of the expression:

```
>>> (x**3 - 3*x).subs({x: oo})
nan
```

```
>>> limit(x**3 - 3*x, x, oo)
oo
```

If the substitution will be followed by numerical evaluation, it is better to pass the substitution to evalf as

```
>>> (1/x).evalf(subs={x: 3.0}, n=21)
0.333333333333333333333
```

rather than

```
>>> (1/x).subs({x: 3.0}).evalf(21)
0.333333333333333314830
```

as the former will ensure that the desired level of precision is obtained.

taylor_term(*n*, *x*, **previous_terms*)

General method for the taylor term.

This method is slow, because it differentiates n-times. Subclasses can redefine it to make it faster by using the "previous_terms".

to_nnf(*simplify=True*)

together(**args*, ***kwargs*)

See the together function in sympy.polys

transpose()

trigsimp(***args*)

See the trigsimp function in sympy.simplify

xreplace(*rule*, *hack2=False*)

Replace occurrences of objects within the expression.

Parameters

rule
[dict-like] Expresses a replacement rule

Returns

xreplace
[the result of the replacement]

See also:

replace
replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

subs
substitution of subexpressions as defined by the objects themselves.

Examples

```
>>> from sympy import symbols, pi, exp
>>> x, y, z = symbols('x y z')
>>> (1 + x*y).xreplace({x: pi})
pi*y + 1
>>> (1 + x*y).xreplace({x: pi, y: 2})
1 + 2*pi
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> (x*y + z).xreplace({x*y: pi})
z + pi
>>> (x*y*z).xreplace({x*y: pi})
x*y*z
>>> (2*x).xreplace({2*x: y, x: z})
y
>>> (2*2*x).xreplace({2*x: y, x: z})
4*z
>>> (x + y + 2).xreplace({x + y: 2})
x + y + 2
>>> (x + 2 + exp(x + 2)).xreplace({x + 2: y})
x + exp(y) + 2
```

xreplace does not differentiate between free and bound symbols. In the following, subs(x, y) would not change x since it is a bound symbol, but xreplace does:

```
>>> from sympy import Integral
>>> Integral(x, (x, 1, 2*x)).xreplace({x: y})
Integral(y, (y, 1, 2*y))
```

Trying to replace x with an expression raises an error:

```
>>> Integral(x, (x, 1, 2*x)).xreplace({x: 2*y})
ValueError: Invalid limits given: ((2*y, 1, 4*y),)
```

58.2.4 Formula

class nipy.algorithms.statistics.formula.formulae.**Formula**(*seq*, *char='b'*)

Bases: `object`

A Formula is a model for a mean in a regression model.

It is often given by a sequence of sympy expressions, with the mean model being the sum of each term multiplied by a linear regression coefficient.

The expressions may depend on additional Symbol instances, giving a non-linear regression model.

__init__(*seq*, *char='b'*)

Parameters

seq
[sequence of `sympy.Basic`]

char
[str, optional] character for regression coefficient

property coefs

Coefficients in the linear regression formula.

design(*input*, *param=None*, *return_float=False*, *contrasts=None*)

Construct the design matrix, and optional contrast matrices.

Parameters

input
[np.recarray] Recarray including fields needed to compute the Terms in get-params(self.design_expr).

param
[None or np.recarray] Recarray including fields that are not Terms in get-params(self.design_expr)

return_float
[bool, optional] If True, return a np.float64 array rather than a np.recarray

contrasts
[None or dict, optional] Contrasts. The items in this dictionary should be (str, Formula) pairs where a contrast matrix is constructed for each Formula by evaluating its design at the same parameters as self.design. If not None, then the return_float is set to True.

Returns

des
[2D array] design matrix

cmatrices
[dict, optional] Dictionary with keys from *contrasts* input, and contrast matrices corre-sponding to *des* design matrix. Returned only if *contrasts* input is not None

property design_expr

property dtype

The dtype of the design matrix of the Formula.

static fromrec(*rec*, *keep=[]*, *drop=[]*)

Construct Formula from recarray

For fields with a string-dtype, it is assumed that these are qualtiatitve regressors, i.e. Factors.

Parameters

rec: recarray
Recarray whose field names will be used to create a formula.

keep: []
Field names to explicitly keep, dropping all others.

drop: []
Field names to drop.

property mean

Expression for the mean, expressed as a linear combination of terms, each with dummy variables in front.

property params

The parameters in the Formula.

subs(*old*, *new*)

Perform a sympy substitution on all terms in the Formula

Returns a new instance of the same class

Parameters

old
[sympy.Basic] The expression to be changed

new
[sympy.Basic] The value to change it to.

Returns

newf
[Formula]

Examples

```
>>> s, t = [Term(l) for l in 'st']
>>> f, g = [sympy.Function(l) for l in 'fg']
>>> form = Formula([f(t),g(s)])
>>> newform = form.subs(g, sympy.Function('h'))
>>> newform.terms
array([f(t), h(s)], dtype=object)
>>> form.terms
array([f(t), g(s)], dtype=object)
```

property terms

Terms in the linear regression formula.

58.2.5 RandomEffects

class nipy.algorithms.statistics.formula.formulae.**RandomEffects**(*seq*, *sigma=None*, *char='e'*)

Bases: *Formula*

Covariance matrices for common random effects analyses.

Examples

Two subjects (here named 2 and 3):

```
>>> subj = make_recarray([2,2,2,3,3], 's')
>>> subj_factor = Factor('s', [2,3])
```

By default the covariance matrix is symbolic. The display differs a little between sympy versions (hence we don't check it in the doctests):

```
>>> c = RandomEffects(subj_factor.terms)
>>> c.cov(subj)
array([[_s2_0, _s2_0, _s2_0, 0, 0],
       [_s2_0, _s2_0, _s2_0, 0, 0],
       [_s2_0, _s2_0, _s2_0, 0, 0],
       [0, 0, 0, _s2_1, _s2_1],
       [0, 0, 0, _s2_1, _s2_1]], dtype=object)
```

With a numeric *sigma*, you get a numeric array:

```
>>> c = RandomEffects(subj_factor.terms, sigma=np.array([[4,1],[1,6]]))
>>> c.cov(subj)
array([[ 4.,  4.,  4.,  1.,  1.],
       [ 4.,  4.,  4.,  1.,  1.],
       [ 4.,  4.,  4.,  1.,  1.],
       [ 1.,  1.,  1.,  6.,  6.],
       [ 1.,  1.,  1.,  6.,  6.]])
```

__init__(*seq*, *sigma=None*, *char='e'*)

Initialize random effects instance

Parameters

seq
[[sympy.Basic]]

sigma
[ndarray] Covariance of the random effects. Defaults to a diagonal with entries for each random effect.

char
[character for regression coefficient]

property coefs

Coefficients in the linear regression formula.

cov(*term*, *param=None*)

Compute the covariance matrix for some given data.

Parameters

term
[np.recarray] Recarray including fields corresponding to the Terms in getparams(self.design_expr).

param
[np.recarray] Recarray including fields that are not Terms in getparams(self.design_expr)

Returns

C
[ndarray] Covariance matrix implied by design and self.sigma.

`design`(*input*, *param=None*, *return_float=False*, *contrasts=None*)

Construct the design matrix, and optional contrast matrices.

Parameters

input
[np.recarray] Recarray including fields needed to compute the Terms in getparams(self.design_expr).

param
[None or np.recarray] Recarray including fields that are not Terms in getparams(self.design_expr)

return_float
[bool, optional] If True, return a np.float64 array rather than a np.recarray

contrasts
[None or dict, optional] Contrasts. The items in this dictionary should be (str, Formula) pairs where a contrast matrix is constructed for each Formula by evaluating its design at the same parameters as self.design. If not None, then the return_float is set to True.

Returns

des
[2D array] design matrix

cmatrices
[dict, optional] Dictionary with keys from *contrasts* input, and contrast matrices corresponding to *des* design matrix. Returned only if *contrasts* input is not None

`property design_expr`

`property dtype`

The dtype of the design matrix of the Formula.

`static fromrec`(*rec*, *keep=[]*, *drop=[]*)

Construct Formula from recarray

For fields with a string-dtype, it is assumed that these are qualtiatitve regressors, i.e. Factors.

Parameters

rec: recarray
Recarray whose field names will be used to create a formula.

keep: []
Field names to explicitly keep, dropping all others.

drop: []
Field names to drop.

property mean

Expression for the mean, expressed as a linear combination of terms, each with dummy variables in front.

property params

The parameters in the Formula.

subs(*old*, *new*)

Perform a sympy substitution on all terms in the Formula

Returns a new instance of the same class

Parameters

old
[sympy.Basic] The expression to be changed

new
[sympy.Basic] The value to change it to.

Returns

newf
[Formula]

Examples

```
>>> s, t = [Term(l) for l in 'st']
>>> f, g = [sympy.Function(l) for l in 'fg']
>>> form = Formula([f(t),g(s)])
>>> newform = form.subs(g, sympy.Function('h'))
>>> newform.terms
array([f(t), h(s)], dtype=object)
>>> form.terms
array([f(t), g(s)], dtype=object)
```

property terms

Terms in the linear regression formula.

58.2.6 Term

class nipy.algorithms.statistics.formula.formulae.**Term**(*name*, ***assumptions*)

Bases: Symbol

A sympy.Symbol type to represent a term an a regression model

Terms can be added to other sympy expressions with the single convention that a term plus itself returns itself.

It is meant to emulate something on the right hand side of a formula in R. In particular, its name can be the name of a field in a recarray used to create a design matrix.

```
>>> t = Term('x')
>>> xval = np.array([(3,),(4,),(5,)], np.dtype([('x', np.float64)]))
>>> f = t.formula
>>> d = f.design(xval)
>>> print(d.dtype.descr)
[('x', '<f8')]
```

(continues on next page)

(continued from previous page)

```
>>> f.design(xval, return_float=True)
array([ 3.,  4.,  5.])
```

__init__(**args*, ***kwargs*)

adjoint()

apart(*x=None*, ***args*)

See the apart function in sympy.polys

property args: tuple[Basic, ...]

Returns a tuple of arguments of 'self'.

Notes

Never use self._args, always use self.args. Only use _args in __new__ when creating a new function. Do not override .args() from Basic (so that it is easy to change the interface in the future if needed).

Examples

```
>>> from sympy import cot
>>> from sympy.abc import x, y
```

```
>>> cot(x).args
(x,)
```

```
>>> cot(x).args[0]
x
```

```
>>> (x*y).args
(x, y)
```

```
>>> (x*y).args[1]
y
```

args_cnc(*cset=False*, *warn=True*, *split_1=True*)

Return [commutative factors, non-commutative factors] of self.

Examples

```
>>> from sympy import symbols, oo
>>> A, B = symbols('A B', commutative=0)
>>> x, y = symbols('x y')
>>> (-2*x*y).args_cnc()
[[-1, 2, x, y], []]
>>> (-2.5*x).args_cnc()
[[-1, 2.5, x], []]
>>> (-2*x*A*B*y).args_cnc()
```

(continues on next page)

(continued from previous page)

```
[[-1, 2, x, y], [A, B]]
>>> (-2*x*A*B*y).args_cnc(split_1=False)
[[-2, x, y], [A, B]]
>>> (-2*x*y).args_cnc(cset=True)
[{-1, 2, x, y}, []]
```

The arg is always treated as a Mul:

```
>>> (-2 + x + A).args_cnc()
[[], [x - 2 + A]]
>>> (-oo).args_cnc() # -oo is a singleton
[[-1, oo], []]
```

as_base_exp() → tuple[Expr, Expr]

as_coeff_Add(*rational=False*) → tuple[Number, Expr]

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational: bool = False*) → tuple[Number, Expr]

Efficiently extract the coefficient of a product.

as_coeff_add(**deps*) → tuple[Expr, tuple[Expr, ...]]

Return the tuple (c, args) where self is written as an Add, `a`.

c should be a Rational added to any terms of the Add that are independent of deps.

args should be a tuple of all other terms of `a`; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you do not know if self is an Add or not but you want to treat self as an Add or if you want to process the individual arguments of the tail of self as an Add.

- if you know self is an Add and want only the head, use self.args[0];
- if you do not want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail.
- if you want to split self into an independent and dependent parts use `self.as_independent(*deps)`

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_add()
(3, ())
>>> (3 + x).as_coeff_add()
(3, (x,))
>>> (3 + x + y).as_coeff_add(x)
(y + 3, (x,))
>>> (3 + y).as_coeff_add(x)
(y + 3, ())
```

as_coeff_exponent(*x*) → tuple[Expr, Expr]

`c*x**e -> c,e` where x can be any symbolic expression.

as_coeff_mul(**deps*, ***kwargs*) → tuple[Expr, tuple[Expr, ...]]

Return the tuple (c, args) where self is written as a Mul, `m`.

c should be a Rational multiplied by any factors of the Mul that are independent of deps.

args should be a tuple of all other factors of m; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you do not know if self is a Mul or not but you want to treat self as a Mul or if you want to process the individual arguments of the tail of self as a Mul.

- if you know self is a Mul and want only the head, use self.args[0];
- if you do not want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail;
- if you want to split self into an independent and dependent parts use `self.as_independent(*deps)`

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_mul()
(3, ())
>>> (3*x*y).as_coeff_mul()
(3, (x, y))
>>> (3*x*y).as_coeff_mul(x)
(3*y, (x,))
>>> (3*y).as_coeff_mul(x)
(3*y, ())
```

`as_coefficient`(*expr*)

Extracts symbolic coefficient at the given expression. In other words, this functions separates 'self' into the product of 'expr' and 'expr'-free coefficient. If such separation is not possible it will return None.

See also:

coeff
return sum of terms have a given factor

as_coeff_Add
separate the additive constant from an expression

as_coeff_Mul
separate the multiplicative constant from an expression

as_independent
separate x-dependent terms/factors from others

sympy.polys.polytools.Poly.coeff_monomial
efficiently find the single coefficient of a monomial in Poly

sympy.polys.polytools.Poly.nth
like coeff_monomial but powers of monomial terms are used

Examples

```
>>> from sympy import E, pi, sin, I, Poly
>>> from sympy.abc import x
```

```
>>> E.as_coefficient(E)
1
>>> (2*E).as_coefficient(E)
```

(continues on next page)

(continued from previous page)

```
2
>>> (2*sin(E)*E).as_coefficient(E)
```

Two terms have E in them so a sum is returned. (If one were desiring the coefficient of the term exactly matching E then the constant from the returned expression could be selected. Or, for greater precision, a method of Poly can be used to indicate the desired term from which the coefficient is desired.)

```
>>> (2*E + x*E).as_coefficient(E)
x + 2
>>> _.args[0]  # just want the exact match
2
>>> p = Poly(2*E + x*E); p
Poly(x*E + 2*E, x, E, domain='ZZ')
>>> p.coeff_monomial(E)
2
>>> p.nth(0, 1)
2
```

Since the following cannot be written as a product containing E as a factor, None is returned. (If the coefficient `2*x` is desired then the `coeff` method should be used.)

```
>>> (2*E*x + x).as_coefficient(E)
>>> (2*E*x + x).coeff(E)
2*x
```

```
>>> (E*(x + 1) + x).as_coefficient(E)
```

```
>>> (2*pi*I).as_coefficient(pi*I)
2
>>> (2*I).as_coefficient(pi*I)
```

as_coefficients_dict(**syms*)

Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0.

If symbols `syms` are provided, any multiplicative terms independent of them will be considered a coefficient and a regular dictionary of syms-dependent generators as keys and their corresponding coefficients as values will be returned.

Examples

```
>>> from sympy.abc import a, x, y
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
>>> (3*a*x).as_coefficients_dict(x)
{x: 3*a}
>>> (3*a*x).as_coefficients_dict(y)
{1: 3*a*x}
```

as_content_primitive(*radical=False*, *clear=True*)

This method should recursively remove a Rational from all arguments and return that (content) and the new self (primitive). The content should always be positive and `Mul(*foo.as_content_primitive()) == foo`. The primitive need not be in canonical form and should try to preserve the underlying structure if possible (i.e. expand_mul should not be applied to self).

Examples

```
>>> from sympy import sqrt
>>> from sympy.abc import x, y, z
```

```
>>> eq = 2 + 2*x + 2*y*(3 + 3*y)
```

The as_content_primitive function is recursive and retains structure:

```
>>> eq.as_content_primitive()
(2, x + 3*y*(y + 1) + 1)
```

Integer powers will have Rationals extracted from the base:

```
>>> ((2 + 6*x)**2).as_content_primitive()
(4, (3*x + 1)**2)
>>> ((2 + 6*x)**(2*y)).as_content_primitive()
(1, (2*(3*x + 1))**(2*y))
```

Terms may end up joining once their as_content_primitives are added:

```
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(11, x*(y + 1))
>>> ((3*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(9, x*(y + 1))
>>> ((3*(z*(1 + y)) + 2.0*x*(3 + 3*y))).as_content_primitive()
(1, 6.0*x*(y + 1) + 3*z*(y + 1))
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))**2).as_content_primitive()
(121, x**2*(y + 1)**2)
>>> ((x*(1 + y) + 0.4*x*(3 + 3*y))**2).as_content_primitive()
(1, 4.84*x**2*(y + 1)**2)
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

If clear=False (default is True) then content will not be removed from an Add if it can be distributed to leave one or more terms with integer coefficients.

```
>>> (x/2 + y).as_content_primitive()
(1/2, x + 2*y)
>>> (x/2 + y).as_content_primitive(clear=False)
(1, x/2 + y)
```

as_dummy()

Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True. When applied to a symbol a new symbol having only the same commutativity will be returned.

Notes

Any object that has structurally bound variables should have a property, *bound_symbols* that returns those symbols appearing in the object.

Examples

```
>>> from sympy import Integral, Symbol
>>> from sympy.abc import x
>>> r = Symbol('r', real=True)
>>> Integral(r, (r, x)).as_dummy()
Integral(_0, (_0, x))
>>> _.variables[0].is_real is None
True
>>> r.as_dummy()
_r
```

as_expr(**gens*)

Convert a polynomial to a SymPy expression.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y
```

```
>>> f = (x**2 + x*y).as_poly(x, y)
>>> f.as_expr()
x**2 + x*y
```

```
>>> sin(x).as_expr()
sin(x)
```

as_independent(**deps*, ***hint*) → tuple[Expr, Expr]

A mostly naive separation of a Mul or Add into arguments that are not are dependent on deps. To obtain as complete a separation of variables as possible, use a separation method first, e.g.:

- separatevars() to change Mul, Add and Pow (including exp) into Mul
- .expand(mul=True) to change Add or Mul into Add
- .expand(log=True) to change log expr into an Add

The only non-naive thing that is done here is to respect noncommutative ordering of variables and to always return (0, 0) for *self* of zero regardless of hints.

For nonzero *self*, the returned tuple (i, d) has the following interpretation:

- i will has no variable that appears in deps
- d will either have terms that contain variables that are in deps, or be equal to 0 (when self is an Add) or 1 (when self is a Mul)

- if self is an Add then self = i + d
- if self is a Mul then self = i*d
- otherwise (self, S.One) or (S.One, self) is returned.

To force the expression to be treated as an Add, use the hint as_Add=True

See also:

separatevars
expand_log
sympy.core.add.Add.as_two_terms
sympy.core.mul.Mul.as_two_terms
as_coeff_mul

Examples

– self is an Add

```
>>> from sympy import sin, cos, exp
>>> from sympy.abc import x, y, z
```

```
>>> (x + x*y).as_independent(x)
(0, x*y + x)
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> (2*x*sin(x) + y + x + z).as_independent(x)
(y + z, 2*x*sin(x) + x)
>>> (2*x*sin(x) + y + x + z).as_independent(x, y)
(z, 2*x*sin(x) + x + y)
```

– self is a Mul

```
>>> (x*sin(x)*cos(y)).as_independent(x)
(cos(y), x*sin(x))
```

non-commutative terms cannot always be separated out when self is a Mul

```
>>> from sympy import symbols
>>> n1, n2, n3 = symbols('n1 n2 n3', commutative=False)
>>> (n1 + n1*n2).as_independent(n2)
(n1, n1*n2)
>>> (n2*n1 + n1*n2).as_independent(n2)
(0, n1*n2 + n2*n1)
>>> (n1*n2*n3).as_independent(n1)
(1, n1*n2*n3)
>>> (n1*n2*n3).as_independent(n2)
(n1, n2*n3)
>>> ((x-n1)*(x-y)).as_independent(x)
(1, (x - y)*(x - n1))
```

– self is anything else:

```
>>> (sin(x)).as_independent(x)
(1, sin(x))
>>> (sin(x)).as_independent(y)
(sin(x), 1)
>>> exp(x+y).as_independent(x)
(1, exp(x + y))
```

– force self to be treated as an Add:

```
>>> (3*x).as_independent(x, as_Add=True)
(0, 3*x)
```

– force self to be treated as a Mul:

```
>>> (3+x).as_independent(x, as_Add=False)
(1, x + 3)
>>> (-3+x).as_independent(x, as_Add=False)
(1, x - 3)
```

Note how the below differs from the above in making the constant on the dep term positive.

```
>>> (y*(-3+x)).as_independent(x)
(y, x - 3)
```

– use .as_independent() for true independence testing instead
of .has(). The former considers only symbols in the free symbols while the latter considers all symbols

```
>>> from sympy import Integral
>>> I = Integral(x, (x, 1, 2))
>>> I.has(x)
True
>>> x in I.free_symbols
False
>>> I.as_independent(x) == (I, 1)
True
>>> (I + x).as_independent(x) == (I, x)
True
```

Note: when trying to get independent terms, a separation method might need to be used first. In this case, it is important to keep track of what you send to this routine so you know how to interpret the returned values

```
>>> from sympy import separatevars, log
>>> separatevars(exp(x+y)).as_independent(x)
(exp(y), exp(x))
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> separatevars(x + x*y).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).expand(mul=True).as_independent(y)
(x, x*y)
>>> a, b=symbols('a b', positive=True)
```

(continues on next page)

(continued from previous page)

```
>>> (log(a*b).expand(log=True)).as_independent(b)
(log(a), log(b))
```

as_leading_term(**symbols*, *logx=None*, *cdir=0*)

Returns the leading (nonzero) term of the series expansion of self.

The _eval_as_leading_term routines are used to do this, and they must always return a non-zero value.

Examples

```
>>> from sympy.abc import x
>>> (1 + x + x**2).as_leading_term(x)
1
>>> (1/x**2 + x + x**2).as_leading_term(x)
x**(-2)
```

as_numer_denom()

Return the numerator and the denominator of an expression.

expression -> a/b -> a, b

This is just a stub that should be defined by an object's class methods to get anything else.

See also:

normal
return a/b instead of (a, b)

as_ordered_factors(*order=None*)

Return list of ordered factors (if Mul) else [self].

as_ordered_terms(*order=None*, *data=False*)

Transform an expression to an ordered list of terms.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.abc import x
```

```
>>> (sin(x)**2*cos(x) + sin(x)**2 + 1).as_ordered_terms()
[sin(x)**2*cos(x), sin(x)**2, 1]
```

as_poly(**gens*, ***args*)

Converts `self` to a polynomial or returns `None`.

as_powers_dict()

Return self as a dictionary of factors with each factor being treated as a power. The keys are the bases of the factors and the values, the corresponding exponents. The resulting dictionary should be used with caution if the expression is a Mul and contains non- commutative factors since the order that they appeared will be lost in the dictionary.

See also:

as_ordered_factors
An alternative for noncommutative applications, returning an ordered list of factors.

args_cnc
Similar to as_ordered_factors, but guarantees separation of commutative and noncommutative factors.

as_real_imag(*deep=True*, ***hints*)

Performs complex expansion on 'self' and returns a tuple containing collected both real and imaginary parts. This method cannot be confused with re() and im() functions, which does not perform complex expansion at evaluation.

However it is possible to expand both re() and im() functions and get exactly the same results as with a single call to this function.

```
>>> from sympy import symbols, I
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> (x + y*I).as_real_imag()
(x, y)
```

```
>>> from sympy.abc import z, w
```

```
>>> (z + w*I).as_real_imag()
(re(z) - im(w), re(w) + im(z))
```

as_set()

Rewrites Boolean expression in terms of real sets.

Examples

```
>>> from sympy import Symbol, Eq, Or, And
>>> x = Symbol('x', real=True)
>>> Eq(x, 0).as_set()
{0}
>>> (x > 0).as_set()
Interval.open(0, oo)
>>> And(-2 < x, x < 2).as_set()
Interval.open(-2, 2)
>>> Or(x < -2, 2 < x).as_set()
Union(Interval.open(-oo, -2), Interval.open(2, oo))
```

as_terms()

Transform an expression to a list of terms.

aseries(*x=None*, *n=6*, *bound=0*, *hir=False*)

Asymptotic Series expansion of self. This is equivalent to `self.series(x, oo, n)`.

Parameters

self
[Expression] The expression whose series is to be expanded.

x
[Symbol] It is the variable of the expression to be calculated.

n
[Value] The value used to represent the order in terms of `x**n`, up to which the series is to be expanded.

hir
[Boolean] Set this parameter to be True to produce hierarchical series. It stops the recursion at an early level and may provide nicer and more useful results.

bound
[Value, Integer] Use the `bound` parameter to give limit on rewriting coefficients in its normalised form.

Returns

Expr
Asymptotic series expansion of the expression.

See also:

`Expr.aseries`
See the docstring of this function for complete details of this wrapper.

Notes

This algorithm is directly induced from the limit computational algorithm provided by Gruntz. It majorly uses the mrv and rewrite sub-routines. The overall idea of this algorithm is first to look for the most rapidly varying subexpression w of a given expression f and then expands f in a series in w. Then same thing is recursively done on the leading coefficient till we get constant coefficients.

If the most rapidly varying subexpression of a given expression f is f itself, the algorithm tries to find a normalised representation of the mrv set and rewrites f using this normalised representation.

If the expansion contains an order term, it will be either `O(x ** (-n))` or `O(w ** (-n))` where `w` belongs to the most rapidly varying expression of `self`.

References

[1], [2], [3]

Examples

```
>>> from sympy import sin, exp
>>> from sympy.abc import x
```

```
>>> e = sin(1/x + exp(-x)) - sin(1/x)
```

```
>>> e.aseries(x)
(1/(24*x**4) - 1/(2*x**2) + 1 + O(x**(-6), (x, oo)))*exp(-x)
```

```
>>> e.aseries(x, n=3, hir=True)
-exp(-2*x)*sin(1/x)/2 + exp(-x)*cos(1/x) + O(exp(-3*x), (x, oo))
```

```
>>> e = exp(exp(x)/(1 - 1/x))
```

```
>>> e.aseries(x)
exp(exp(x)/(1 - 1/x))
```

```
>>> e.aseries(x, bound=3)
exp(exp(x)/x**2)*exp(exp(x)/x)*exp(-exp(x) + exp(x)/(1 - 1/x) - exp(x)/x -␣
↪exp(x)/x**2)*exp(exp(x))
```

For rational expressions this method may return original expression without the Order term. >>> (1/x).aseries(x, n=8) 1/x

property assumptions0

Return object *type* assumptions.

For example:

Symbol('x', real=True) Symbol('x', integer=True)

are different objects. In other words, besides Python type (Symbol in this case), the initial assumptions are also forming their typeinfo.

Examples

```
>>> from sympy import Symbol
>>> from sympy.abc import x
>>> x.assumptions0
{'commutative': True}
>>> x = Symbol("x", positive=True)
>>> x.assumptions0
{'commutative': True, 'complex': True, 'extended_negative': False,
 'extended_nonnegative': True, 'extended_nonpositive': False,
 'extended_nonzero': True, 'extended_positive': True, 'extended_real':
 True, 'finite': True, 'hermitian': True, 'imaginary': False,
 'infinite': False, 'negative': False, 'nonnegative': True,
 'nonpositive': False, 'nonzero': True, 'positive': True, 'real':
 True, 'zero': False}
```

atoms(**types*)

Returns the atoms that form the current object.

By default, only objects that are truly atomic and cannot be divided into smaller pieces are returned: symbols, numbers, and number symbols like I and pi. It is possible to request atoms of any type, however, as demonstrated below.

Examples

```
>>> from sympy import I, pi, sin
>>> from sympy.abc import x, y
>>> (1 + x + 2*sin(y + I*pi)).atoms()
{1, 2, I, pi, x, y}
```

If one or more types are given, the results will contain only those types of atoms.

```
>>> from sympy import Number, NumberSymbol, Symbol
>>> (1 + x + 2*sin(y + I*pi)).atoms(Symbol)
{x, y}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number)
{1, 2}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol)
{1, 2, pi}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol, I)
{1, 2, I, pi}
```

Note that I (imaginary unit) and zoo (complex infinity) are special types of number symbols and are not part of the NumberSymbol class.

The type can be given implicitly, too:

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(x) # x is a Symbol
{x, y}
```

Be careful to check your assumptions when using the implicit option since `S(1).is_Integer = True` but `type(S(1))` is `One`, a special type of SymPy atom, while `type(S(2))` is type `Integer` and will find all integers in an expression:

```
>>> from sympy import S
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(1))
{1}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(2))
{1, 2}
```

Finally, arguments to atoms() can select more than atomic atoms: any SymPy type (loaded in core/__init__.py) can be listed as an argument and those types of "atoms" as found in scanning the arguments of the expression recursively:

```
>>> from sympy import Function, Mul
>>> from sympy.core.function import AppliedUndef
>>> f = Function('f')
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(Function)
{f(x), sin(y + I*pi)}
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(AppliedUndef)
{f(x)}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
{I*pi, 2*sin(y + I*pi)}
```

property binary_symbols

Return from the atoms of self those which are free symbols.

Not all free symbols are `Symbol`. Eg: IndexedBase('I')[0].free_symbols

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

cancel(**gens*, ***args*)

See the cancel function in sympy.polys

property canonical_variables

Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

Examples

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> Lambda(x, 2*x).canonical_variables
{x: _0}
```

classmethod class_key()

Nice order of classes.

coeff(*x*, *n=1*, *right=False*, *_first=True*)

Returns the coefficient from the term(s) containing `x**n`. If `n` is zero then all terms independent of `x` will be returned.

See also:

as_coefficient
separate the expression into a coefficient and factor

as_coeff_Add
separate the additive constant from an expression

as_coeff_Mul
separate the multiplicative constant from an expression

as_independent
separate x-dependent terms/factors from others

sympy.polys.polytools.Poly.coeff_monomial
efficiently find the single coefficient of a monomial in Poly

sympy.polys.polytools.Poly.nth
like coeff_monomial but powers of monomial terms are used

Examples

```
>>> from sympy import symbols
>>> from sympy.abc import x, y, z
```

You can select terms that have an explicit negative in front of them:

```
>>> (-x + 2*y).coeff(-1)
x
>>> (x - 2*y).coeff(-1)
2*y
```

You can select terms with no Rational coefficient:

```
>>> (x + 2*y).coeff(1)
x
>>> (3 + 2*x + 4*x**2).coeff(1)
0
```

You can select terms independent of x by making n=0; in this case expr.as_independent(x)[0] is returned (and 0 will be returned instead of None):

```
>>> (3 + 2*x + 4*x**2).coeff(x, 0)
3
>>> eq = ((x + 1)**3).expand() + 1
>>> eq
x**3 + 3*x**2 + 3*x + 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 2]
>>> eq -= 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 0]
```

You can select terms that have a numerical term in front of them:

```
>>> (-x - 2*y).coeff(2)
-y
>>> from sympy import sqrt
>>> (x + sqrt(2)*x).coeff(sqrt(2))
x
```

The matching is exact:

```
>>> (3 + 2*x + 4*x**2).coeff(x)
2
>>> (3 + 2*x + 4*x**2).coeff(x**2)
4
>>> (3 + 2*x + 4*x**2).coeff(x**3)
0
>>> (z*(x + y)**2).coeff((x + y)**2)
z
>>> (z*(x + y)**2).coeff(x + y)
0
```

In addition, no factoring is done, so 1 + z*(1 + y) is not obtained from the following:

```
>>> (x + z*(x + x*y)).coeff(x)
1
```

If such factoring is desired, factor_terms can be used first:

```
>>> from sympy import factor_terms
>>> factor_terms(x + z*(x + x*y)).coeff(x)
z*(y + 1) + 1
```

```
>>> n, m, o = symbols('n m o', commutative=False)
>>> n.coeff(n)
1
>>> (3*n).coeff(n)
3
>>> (n*m + m*n*m).coeff(n) # = (1 + m)*n*m
1 + m
>>> (n*m + m*n*m).coeff(n, right=True) # = (1 + m)*n*m
m
```

If there is more than one possible coefficient 0 is returned:

```
>>> (n*m + m*n).coeff(n)
0
```

If there is only one possible coefficient, it is returned:

```
>>> (n*m + x*m*n).coeff(m*n)
x
>>> (n*m + x*m*n).coeff(m*n, right=1)
1
```

collect(*syms*, *func=None*, *evaluate=True*, *exact=False*, *distribute_order_term=True*)

See the collect function in sympy.simplify

combsimp()

See the combsimp function in sympy.simplify

compare(*other*)

Return -1, 0, 1 if the object is smaller, equal, or greater than other.

Not in the mathematical sense. If the object is of a different type from the "other" then their classes are ordered according to the sorted_classes list.

Examples

```
>>> from sympy.abc import x, y
>>> x.compare(y)
-1
>>> x.compare(x)
0
>>> y.compare(x)
1
```

compute_leading_term(*x*, *logx=None*)

Deprecated function to compute the leading term of a series.

as_leading_term is only allowed for results of .series() This is a wrapper to compute a series first.

conjugate()

Returns the complex conjugate of 'self'.

copy()

could_extract_minus_sign()

Return True if self has -1 as a leading factor or has more literal negative signs than positive signs in a sum, otherwise False.

Examples

```
>>> from sympy.abc import x, y
>>> e = x - y
>>> {i.could_extract_minus_sign() for i in (e, -e)}
{False, True}
```

Though the `y - x` is considered like `-(x - y)`, since it is in a product without a leading factor of -1, the result is false below:

```
>>> (x*(y - x)).could_extract_minus_sign()
False
```

To put something in canonical form wrt to sign, use *signsimp*:

```
>>> from sympy import signsimp
>>> signsimp(x*(y - x))
-x*(x - y)
>>> _.could_extract_minus_sign()
True
```

count(*query*)

Count the number of matching subexpressions.

count_ops(*visual=None*)

Wrapper for count_ops that returns the operation count.

default_assumptions = {}

diff(**symbols*, ***assumptions*)

dir(*x*, *cdir*)

doit(***hints*)

Evaluate objects that are not evaluated by default like limits, integrals, sums and products. All objects of this kind will be evaluated recursively, unless some species were excluded via 'hints' or unless the 'deep' hint was set to 'False'.

```
>>> from sympy import Integral
>>> from sympy.abc import x
>>> 2*Integral(x, x)
2*Integral(x, x)
```

```
>>> (2*Integral(x, x)).doit()
x**2
```

```
>>> (2*Integral(x, x)).doit(deep=False)
2*Integral(x, x)
```

dummy_eq(*other*, *symbol=None*)

Compare two expressions and handle dummy symbols.

Examples

```
>>> from sympy import Dummy
>>> from sympy.abc import x, y
```

```
>>> u = Dummy('u')
```

```
>>> (u**2 + 1).dummy_eq(x**2 + 1)
True
>>> (u**2 + 1) == (x**2 + 1)
False
```

```
>>> (u**2 + y).dummy_eq(x**2 + y, x)
True
>>> (u**2 + y).dummy_eq(x**2 + y, y)
False
```

equals(*other*, *failing_expression=False*)

Return True if self == other, False if it does not, or None. If failing_expression is True then the expression which did not simplify to a 0 will be returned instead of None.

evalf(*n=15*, *subs=None*, *maxn=100*, *chop=False*, *strict=False*, *quad=None*, *verbose=False*)

Evaluate the given formula to an accuracy of *n* digits.

Parameters

subs
[dict, optional] Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

maxn
[int, optional] Allow a maximum temporary working precision of maxn digits.

chop
[bool or number, optional] Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When `True` the chop value defaults to standard precision.

Otherwise the chop value is used to determine the magnitude of "small" for purposes of chopping.

```
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
0.000100000000000000
>>> N(x, chop=1e-5)
0.000100000000000000
>>> N(x, chop=1e-4)
0
```

strict
[bool, optional] Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available maxprec.

quad
[str, optional] Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

verbose
[bool, optional] Print debug information.

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

expand(*deep=True*, *modulus=None*, *power_base=True*, *power_exp=True*, *mul=True*, *log=True*, *multinomial=True*, *basic=True*, ***hints*)

Expand an expression using hints.

See the docstring of the expand() function in sympy.core.function for more information.

property expr_free_symbols

Like `free_symbols`, but returns the free symbols only if they are contained in an expression node.

Examples

```
>>> from sympy.abc import x, y
>>> (x + y).expr_free_symbols
{x, y}
```

If the expression is contained in a non-expression object, do not return the free symbols. Compare:

```
>>> from sympy import Tuple
>>> t = Tuple(x + y)
>>> t.expr_free_symbols
set()
>>> t.free_symbols
{x, y}
```

extract_additively(*c*)

Return self - c if it's possible to subtract c from self and make all matching coefficients move towards zero, else return None.

See also:

extract_multiplicatively
coeff
as_coefficient

Examples

```
>>> from sympy.abc import x, y
>>> e = 2*x + 3
>>> e.extract_additively(x + 1)
x + 2
>>> e.extract_additively(3*x)
>>> e.extract_additively(4)
>>> (y*(x + 1)).extract_additively(x + 1)
>>> ((x + 1)*(x + 2*y + 1) + 3).extract_additively(x + 1)
(x + 1)*(x + 2*y) + 3
```

extract_branch_factor(*allow_half=False*)

Try to write self as `exp_polar(2*pi*I*n)*z` in a nice way. Return (z, n).

```
>>> from sympy import exp_polar, I, pi
>>> from sympy.abc import x, y
>>> exp_polar(I*pi).extract_branch_factor()
(exp_polar(I*pi), 0)
>>> exp_polar(2*I*pi).extract_branch_factor()
(1, 1)
>>> exp_polar(-pi*I).extract_branch_factor()
(exp_polar(I*pi), -1)
>>> exp_polar(3*pi*I + x).extract_branch_factor()
(exp_polar(x + I*pi), 1)
>>> (y*exp_polar(-5*pi*I)*exp_polar(3*pi*I + 2*pi*x)).extract_branch_factor()
(y*exp_polar(2*pi*x), -1)
```

(continues on next page)

(continued from previous page)

```
>>> exp_polar(-I*pi/2).extract_branch_factor()
(exp_polar(-I*pi/2), 0)
```

If allow_half is True, also extract exp_polar(I*pi):

```
>>> exp_polar(I*pi).extract_branch_factor(allow_half=True)
(1, 1/2)
>>> exp_polar(2*I*pi).extract_branch_factor(allow_half=True)
(1, 1)
>>> exp_polar(3*I*pi).extract_branch_factor(allow_half=True)
(1, 3/2)
>>> exp_polar(-I*pi).extract_branch_factor(allow_half=True)
(1, -1/2)
```

extract_multiplicatively(*c*)

Return None if it's not possible to make self in the form c * something in a nice way, i.e. preserving the properties of arguments of self.

Examples

```
>>> from sympy import symbols, Rational
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> ((x*y)**3).extract_multiplicatively(x**2 * y)
x*y**2
```

```
>>> ((x*y)**3).extract_multiplicatively(x**4 * y)
```

```
>>> (2*x).extract_multiplicatively(2)
x
```

```
>>> (2*x).extract_multiplicatively(3)
```

```
>>> (Rational(1, 2)*x).extract_multiplicatively(3)
x/6
```

factor(**gens*, ***args*)

See the factor() function in sympy.polys.polytools

find(*query*, *group=False*)

Find all subexpressions matching a query.

property formula

Return a Formula with only terms=[self].

fourier_series(*limits=None*)

Compute fourier sine/cosine series of self.

See the docstring of the *fourier_series()* in sympy.series.fourier for more information.

fps(*x=None*, *x0=0*, *dir=1*, *hyper=True*, *order=4*, *rational=True*, *full=False*)

Compute formal power power series of self.

See the docstring of the *fps()* function in sympy.series.formal for more information.

property free_symbols

Return from the atoms of self those which are free symbols.

Not all free symbols are Symbol. Eg: IndexedBase('I')[0].free_symbols

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

classmethod fromiter(*args*, ***assumptions*)

Create a new object from an iterable.

This is a convenience function that allows one to create objects from any iterable, without having to convert to a list or tuple first.

Examples

```
>>> from sympy import Tuple
>>> Tuple.fromiter(i for i in range(5))
(0, 1, 2, 3, 4)
```

property func

The top-level function in an expression.

The following should hold for all objects:

```
>> x == x.func(*x.args)
```

Examples

```
>>> from sympy.abc import x
>>> a = 2*x
>>> a.func
<class 'sympy.core.mul.Mul'>
>>> a.args
(2, x)
>>> a.func(*a.args)
2*x
>>> a == a.func(*a.args)
True
```

gammasimp()

See the gammasimp function in sympy.simplify

getO()

Returns the additive O(..) symbol if there is one, else None.

getn()

Returns the order of the expression.

Examples

```
>>> from sympy import O
>>> from sympy.abc import x
>>> (1 + x + O(x**2)).getn()
2
>>> (1 + x).getn()
```

has(**patterns*)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y, z
>>> (x**2 + sin(x*y)).has(z)
False
>>> (x**2 + sin(x*y)).has(x, y, z)
True
>>> x.has(x)
True
```

Note `has` is a structural algorithm with no knowledge of mathematics. Consider the following half-open interval:

```
>>> from sympy import Interval
>>> i = Interval.Lopen(0, 5); i
Interval.Lopen(0, 5)
>>> i.args
(0, 5, True, False)
>>> i.has(4)  # there is no "4" in the arguments
False
>>> i.has(0)  # there *is* a "0" in the arguments
True
```

Instead, use `contains` to determine whether a number is in the interval or not:

```
>>> i.contains(4)
True
>>> i.contains(0)
False
```

Note that `expr.has(*patterns)` is exactly equivalent to `any(expr.has(p) for p in patterns)`. In particular, `False` is returned when the list of patterns is empty.

```
>>> x.has()
False
```

has_free(**patterns*)

Return True if self has object(s) `x` as a free expression else False.

Examples

```
>>> from sympy import Integral, Function
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> g = Function('g')
>>> expr = Integral(f(x), (f(x), 1, g(y)))
>>> expr.free_symbols
{y}
>>> expr.has_free(g(y))
True
>>> expr.has_free(*(x, f(x)))
False
```

This works for subexpressions and types, too:

```
>>> expr.has_free(g)
True
>>> (x + y + 1).has_free(y + 1)
True
```

has_xfree(*s: set[Basic]*)

Return True if self has any of the patterns in s as a free argument, else False. This is like *Basic.has_free* but this will only report exact argument matches.

Examples

```
>>> from sympy import Function
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> f(x).has_xfree({f})
False
>>> f(x).has_xfree({f(x)})
True
>>> f(x + 1).has_xfree({x})
True
>>> f(x + 1).has_xfree({x + 1})
True
>>> f(x + y + 1).has_xfree({x + 1})
False
```

integrate(**args*, ***kwargs*)

See the integrate function in sympy.integrals

invert(*g*, **gens*, ***args*)

Return the multiplicative inverse of `self` mod `g` where `self` (and `g`) may be symbolic expressions).

See also:

sympy.core.numbers.mod_inverse, sympy.polys.polytools.invert

is_Add = False

is_AlgebraicNumber = False

is_Atom = True

is_Boolean = False

is_Derivative = False

is_Dummy = False

is_Equality = False

is_Float = False

is_Function = False

is_Indexed = False

is_Integer = False

is_MatAdd = False

is_MatMul = False

is_Matrix = False

is_Mul = False

is_Not = False

is_Number = False

is_NumberSymbol = False

is_Order = False

is_Piecewise = False

is_Point = False

is_Poly = False

is_Pow = False

is_Rational = False

is_Relational = False

is_Symbol = True

is_Vector = False

is_Wild = False

property is_algebraic

is_algebraic_expr(**syms*)

This tests whether a given expression is algebraic or not, in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are "algebraic expressions" with symbolic exponents. This is a simple extension to the is_rational_function, including rational exponentiation.

See also:

is_rational_function

References

[1]

Examples

```
>>> from sympy import Symbol, sqrt
>>> x = Symbol('x', real=True)
>>> sqrt(1 + x).is_rational_function()
False
>>> sqrt(1 + x).is_algebraic_expr()
True
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be an algebraic expression to become one.

```
>>> from sympy import exp, factor
>>> a = sqrt(exp(x)**2 + 2*exp(x) + 1)/(exp(x) + 1)
>>> a.is_algebraic_expr(x)
False
>>> factor(a).is_algebraic_expr()
True
```

property is_antihermitian

property is_commutative

is_comparable = False

property is_complex

property is_composite

is_constant(**wrt*, ***flags*)

Return True if self is constant, False if not, or None if the constancy could not be determined conclusively.

Examples

```
>>> from sympy import cos, sin, Sum, S, pi
>>> from sympy.abc import a, n, x, y
>>> x.is_constant()
False
>>> S(2).is_constant()
True
>>> Sum(x, (x, 1, 10)).is_constant()
True
>>> Sum(x, (x, 1, n)).is_constant()
False
>>> Sum(x, (x, 1, n)).is_constant(y)
True
>>> Sum(x, (x, 1, n)).is_constant(n)
False
>>> Sum(x, (x, 1, n)).is_constant(x)
True
>>> eq = a*cos(x)**2 + a*sin(x)**2 - a
>>> eq.is_constant()
True
>>> eq.subs({x: pi, a: 2}) == eq.subs({x: pi, a: 3}) == 0
True
```

```
>>> (0**x).is_constant()
False
>>> x.is_constant()
False
>>> (x**x).is_constant()
False
>>> one = cos(x)**2 + sin(x)**2
>>> one.is_constant()
True
>>> ((one - 1)**(x + 1)).is_constant() in (True, False) # could be 0 or 1
True
```

property is_even

property is_extended_negative

property is_extended_nonnegative

property is_extended_nonpositive

property is_extended_nonzero

property is_extended_positive

property is_extended_real

property is_finite

property is_hermitian

is_hypergeometric(*k*)

property is_imaginary

property is_infinite

property is_integer

property is_irrational

is_meromorphic(*x*, *a*)

This tests whether an expression is meromorphic as a function of the given symbol x at the point a.

This method is intended as a quick test that will return None if no decision can be made without simplification or more detailed analysis.

Examples

```
>>> from sympy import zoo, log, sin, sqrt
>>> from sympy.abc import x
```

```
>>> f = 1/x**2 + 1 - 2*x**3
>>> f.is_meromorphic(x, 0)
True
>>> f.is_meromorphic(x, 1)
True
>>> f.is_meromorphic(x, zoo)
True
```

```
>>> g = x**log(3)
>>> g.is_meromorphic(x, 0)
False
>>> g.is_meromorphic(x, 1)
True
>>> g.is_meromorphic(x, zoo)
False
```

```
>>> h = sin(1/x)*x**2
>>> h.is_meromorphic(x, 0)
False
>>> h.is_meromorphic(x, 1)
True
>>> h.is_meromorphic(x, zoo)
True
```

Multivalued functions are considered meromorphic when their branches are meromorphic. Thus most functions are meromorphic everywhere except at essential singularities and branch points. In particular, they will be meromorphic also on branch cuts except at their endpoints.

```
>>> log(x).is_meromorphic(x, -1)
True
>>> log(x).is_meromorphic(x, 0)
False
>>> sqrt(x).is_meromorphic(x, -1)
True
```

(continues on next page)

(continued from previous page)

```
>>> sqrt(x).is_meromorphic(x, 0)
False
```

property is_negative

property is_noninteger

property is_nonnegative

property is_nonpositive

property is_nonzero

is_number = False

property is_odd

property is_polar

is_polynomial(**syms*)

Return True if self is a polynomial in syms and False otherwise.

This checks if self is an exact polynomial in syms. This function returns False for expressions that are "polynomials" with symbolic exponents. Thus, you should be able to apply polynomial algorithms to expressions for which this returns True, and Poly(expr, *syms) should work if and only if expr.is_polynomial(*syms) returns True. The polynomial does not have to be in expanded form. If no symbols are given, all free symbols in the expression will be used.

This is not part of the assumptions system. You cannot do Symbol('z', polynomial=True).

Examples

```
>>> from sympy import Symbol, Function
>>> x = Symbol('x')
>>> ((x**2 + 1)**4).is_polynomial(x)
True
>>> ((x**2 + 1)**4).is_polynomial()
True
>>> (2**x + 1).is_polynomial(x)
False
>>> (2**x + 1).is_polynomial(2**x)
True
>>> f = Function('f')
>>> (f(x) + 1).is_polynomial(x)
False
>>> (f(x) + 1).is_polynomial(f(x))
True
>>> (1/f(x) + 1).is_polynomial(f(x))
False
```

```
>>> n = Symbol('n', nonnegative=True, integer=True)
>>> (x**n + 1).is_polynomial(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a polynomial to become one.

```
>>> from sympy import sqrt, factor, cancel
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)
>>> a.is_polynomial(y)
False
>>> factor(a)
y + 1
>>> factor(a).is_polynomial(y)
True
```

```
>>> b = (y**2 + 2*y + 1)/(y + 1)
>>> b.is_polynomial(y)
False
>>> cancel(b)
y + 1
>>> cancel(b).is_polynomial(y)
True
```

See also .is_rational_function()

property is_positive

property is_prime

property is_rational

is_rational_function(**syms*)

Test whether function is a ratio of two polynomials in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are "rational functions" with symbolic exponents. Thus, you should be able to call .as_numer_denom() and apply polynomial algorithms to the result for expressions for which this returns True.

This is not part of the assumptions system. You cannot do Symbol('z', rational_function=True).

Examples

```
>>> from sympy import Symbol, sin
>>> from sympy.abc import x, y
```

```
>>> (x/y).is_rational_function()
True
```

```
>>> (x**2).is_rational_function()
True
```

```
>>> (x/sin(y)).is_rational_function(y)
False
```

```
>>> n = Symbol('n', integer=True)
>>> (x**n + 1).is_rational_function(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a rational function to become one.

```
>>> from sympy import sqrt, factor
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)/y
>>> a.is_rational_function(y)
False
>>> factor(a)
(y + 1)/y
>>> factor(a).is_rational_function(y)
True
```

See also is_algebraic_expr().

property is_real

is_scalar = True

is_symbol = True

property is_transcendental

property is_zero

property kind

Default kind for all SymPy object. If the kind is not defined for the object, or if the object cannot infer the kind from its arguments, this will be returned.

Examples

```
>>> from sympy import Expr
>>> Expr().kind
UndefinedKind
```

leadterm(*x*, *logx=None*, *cdir=0*)

Returns the leading term a*x**b as a tuple (a, b).

Examples

```
>>> from sympy.abc import x
>>> (1+x+x**2).leadterm(x)
(1, 0)
>>> (1/x**2+x+x**2).leadterm(x)
(1, -2)
```

limit(*x*, *xlim*, *dir='+'*)

Compute limit x->xlim.

lseries(*x=None*, *x0=0*, *dir='+'*, *logx=None*, *cdir=0*)

Wrapper for series yielding an iterator of the terms of the series.

Note: an infinite series will yield an infinite iterator. The following, for exaxmple, will never terminate. It will just keep printing terms of the sin(x) series:

```
for term in sin(x).lseries(x):
    print term
```

The advantage of lseries() over nseries() is that many times you are just interested in the next term in the series (i.e. the first term for example), but you do not know how many you should ask for in nseries() using the "n" parameter.

See also nseries().

match(*pattern*, *old=False*)

Pattern matching.

Wild symbols match all.

Return `None` when expression (self) does not match with pattern. Otherwise return a dictionary such that:

```
pattern.xreplace(self.match(pattern)) == self
```

Examples

```
>>> from sympy import Wild, Sum
>>> from sympy.abc import x, y
>>> p = Wild("p")
>>> q = Wild("q")
>>> r = Wild("r")
>>> e = (x+y)**(x+y)
>>> e.match(p**p)
{p_: x + y}
>>> e.match(p**q)
{p_: x + y, q_: x + y}
>>> e = (2*x)**2
>>> e.match(p*q**r)
{p_: 4, q_: x, r_: 2}
>>> (p*q**r).xreplace(e.match(p*q**r))
4*x**2
```

Structurally bound symbols are ignored during matching:

```
>>> Sum(x, (x, 1, 2)).match(Sum(y, (y, 1, p)))
{p_: 2}
```

But they can be identified if desired:

```
>>> Sum(x, (x, 1, 2)).match(Sum(q, (q, 1, p)))
{p_: 2, q_: x}
```

The `old` flag will give the old-style pattern matching where expressions and patterns are essentially solved to give the match. Both of the following give None unless `old=True`:

```
>>> (x - 2).match(p - x, old=True)
{p_: 2*x - 2}
>>> (2/x).match(p*x, old=True)
{p_: 2/x**2}
```

matches(*expr*, *repl_dict=None*, *old=False*)

Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.

Examples

```
>>> from sympy import symbols, Wild, Basic
>>> a, b, c = symbols('a b c')
>>> x = Wild('x')
>>> Basic(a + x, x).matches(Basic(a + b, c)) is None
True
>>> Basic(a + x, x).matches(Basic(a + b + c, b + c))
{x_: b + c}
```

n(*n=15*, *subs=None*, *maxn=100*, *chop=False*, *strict=False*, *quad=None*, *verbose=False*)

Evaluate the given formula to an accuracy of *n* digits.

Parameters

subs
[dict, optional] Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

maxn
[int, optional] Allow a maximum temporary working precision of maxn digits.

chop
[bool or number, optional] Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When `True` the chop value defaults to standard precision.

Otherwise the chop value is used to determine the magnitude of "small" for purposes of chopping.

```
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
0.000100000000000000
>>> N(x, chop=1e-5)
0.000100000000000000
>>> N(x, chop=1e-4)
0
```

strict
[bool, optional] Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available maxprec.

quad
[str, optional] Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

verbose
[bool, optional] Print debug information.

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

name: str

normal()

Return the expression as a fraction.

expression -> a/b

See also:

as_numer_denom
return (a, b) instead of a/b

nseries(*x=None*, *x0=0*, *n=6*, *dir='+'*, *logx=None*, *cdir=0*)

Wrapper to _eval_nseries if assumptions allow, else to series.

If x is given, x0 is 0, dir='+', and self has x, then _eval_nseries is called. This calculates "n" terms in the innermost expressions and then builds up the final series just by "cross-multiplying" everything out.

The optional `logx` parameter can be used to replace any log(x) in the returned series with a symbolic value to avoid evaluating log(x) at 0. A symbol to use in place of log(x) should be provided.

Advantage – it's fast, because we do not have to determine how many terms we need to calculate in advance.

Disadvantage – you may end up with less terms than you may have expected, but the O(x**n) term appended will always be correct and so the result, though perhaps shorter, will also be correct.

If any of those assumptions is not met, this is treated like a wrapper to series which will try harder to return the correct number of terms.

See also lseries().

Examples

```
>>> from sympy import sin, log, Symbol
>>> from sympy.abc import x, y
>>> sin(x).nseries(x, 0, 6)
x - x**3/6 + x**5/120 + O(x**6)
>>> log(x+1).nseries(x, 0, 5)
x - x**2/2 + x**3/3 - x**4/4 + O(x**5)
```

Handling of the `logx` parameter — in the following example the expansion fails since `sin` does not have an asymptotic expansion at -oo (the limit of log(x) as x approaches 0):

```
>>> e = sin(log(x))
>>> e.nseries(x, 0, 6)
Traceback (most recent call last):
...
PoleError: ...
...
>>> logx = Symbol('logx')
>>> e.nseries(x, 0, 6, logx=logx)
sin(logx)
```

In the following example, the expansion works but only returns self unless the `logx` parameter is used:

```
>>> e = x**y
>>> e.nseries(x, 0, 2)
x**y
>>> e.nseries(x, 0, 2, logx=logx)
exp(logx*y)
```

nsimplify(*constants=()*, *tolerance=None*, *full=False*)

See the nsimplify function in sympy.simplify

powsimp(**args*, ***kwargs*)

See the powsimp function in sympy.simplify

primitive()

Return the positive Rational that can be extracted non-recursively from every term of self (i.e., self is treated like an Add). This is like the as_coeff_Mul() method but primitive always extracts a positive Rational (never a negative or a Float).

Examples

```
>>> from sympy.abc import x
>>> (3*(x + 1)**2).primitive()
(3, (x + 1)**2)
>>> a = (6*x + 2); a.primitive()
(2, 3*x + 1)
>>> b = (x/2 + 3); b.primitive()
(1/2, x + 6)
>>> (a*b).primitive() == (1, a*b)
True
```

radsimp(***kwargs*)

See the radsimp function in sympy.simplify

ratsimp()

See the ratsimp function in sympy.simplify

rcall(**args*)

Apply on the argument recursively through the expression tree.

This method is used to simulate a common abuse of notation for operators. For instance, in SymPy the following will not work:

`(x+Lambda(y, 2*y))(z) == x+2*z`,

however, you can use:

```
>>> from sympy import Lambda
>>> from sympy.abc import x, y, z
>>> (x + Lambda(y, 2*y)).rcall(z)
x + 2*z
```

refine(*assumption=True*)

See the refine function in sympy.assumptions

removeO()

Removes the additive O(..) symbol if there is one

replace(*query*, *value*, *map=False*, *simultaneous=True*, *exact=None*)

Replace matching subexpressions of `self` with `value`.

If `map = True` then also return the mapping {old: new} where `old` was a sub-expression found with query and `new` is the replacement value for it. If the expression itself does not match the query, then the returned value will be `self.xreplace(map)` otherwise it should be `self.subs(ordered(map.items()))`.

Traverses an expression tree and performs replacement of matching subexpressions from the bottom to the top of the tree. The default approach is to do the replacement in a simultaneous fashion so changes made are targeted only once. If this is not desired or causes problems, `simultaneous` can be set to False.

In addition, if an expression containing more than one Wild symbol is being used to match subexpressions and the `exact` flag is None it will be set to True so the match will only succeed if all non-zero values are received for each Wild that appears in the match pattern. Setting this to False accepts a match of 0; while setting it True accepts all matches that have a 0 in them. See example below for cautions.

The list of possible combinations of queries and replacement values is listed below:

See also:

`subs`
substitution of subexpressions as defined by the objects themselves.

`xreplace`
exact node replacement in expr tree; also capable of using matching rules

Examples

Initial setup

```
>>> from sympy import log, sin, cos, tan, Wild, Mul, Add
>>> from sympy.abc import x, y
>>> f = log(sin(x)) + tan(sin(x**2))
```

1.1. type -> type
obj.replace(type, newtype)

When object of type `type` is found, replace it with the result of passing its argument(s) to `newtype`.

```
>>> f.replace(sin, cos)
log(cos(x)) + tan(cos(x**2))
>>> sin(x).replace(sin, cos, map=True)
(cos(x), {sin(x): cos(x)})
>>> (x*y).replace(Mul, Add)
x + y
```

1.2. type -> func
obj.replace(type, func)

When object of type `type` is found, apply `func` to its argument(s). `func` must be written to handle the number of arguments of `type`.

```
>>> f.replace(sin, lambda arg: sin(2*arg))
log(sin(2*x)) + tan(sin(2*x**2))
>>> (x*y).replace(Mul, lambda *args: sin(2*Mul(*args)))
sin(2*x*y)
```

2.1. pattern -> expr
obj.replace(pattern(wild), expr(wild))

Replace subexpressions matching `pattern` with the expression written in terms of the Wild symbols in `pattern`.

```
>>> a, b = map(Wild, 'ab')
>>> f.replace(sin(a), tan(a))
log(tan(x)) + tan(tan(x**2))
>>> f.replace(sin(a), tan(a/2))
log(tan(x/2)) + tan(tan(x**2/2))
>>> f.replace(sin(a), a)
log(x) + tan(x**2)
>>> (x*y).replace(a*x, a)
y
```

Matching is exact by default when more than one Wild symbol is used: matching fails unless the match gives non-zero values for all Wild symbols:

```
>>> (2*x + y).replace(a*x + b, b - a)
y - 2
>>> (2*x).replace(a*x + b, b - a)
2*x
```

When set to False, the results may be non-intuitive:

```
>>> (2*x).replace(a*x + b, b - a, exact=False)
2/x
```

2.2. pattern -> func
obj.replace(pattern(wild), lambda wild: expr(wild))

All behavior is the same as in 2.1 but now a function in terms of pattern variables is used rather than an expression:

```
>>> f.replace(sin(a), lambda a: sin(2*a))
log(sin(2*x)) + tan(sin(2*x**2))
```

3.1. func -> func
obj.replace(filter, func)

Replace subexpression `e` with `func(e)` if `filter(e)` is True.

```
>>> g = 2*sin(x**3)
>>> g.replace(lambda expr: expr.is_Number, lambda expr: expr**2)
4*sin(x**9)
```

The expression itself is also targeted by the query but is done in such a fashion that changes are not made twice.

```
>>> e = x*(x*y + 1)
>>> e.replace(lambda x: x.is_Mul, lambda x: 2*x)
2*x*(2*x*y + 1)
```

When matching a single symbol, *exact* will default to True, but this may or may not be the behavior that is desired:

Here, we want *exact=False*:

```
>>> from sympy import Function
>>> f = Function('f')
>>> e = f(1) + f(0)
>>> q = f(a), lambda a: f(a + 1)
>>> e.replace(*q, exact=False)
f(1) + f(2)
>>> e.replace(*q, exact=True)
f(0) + f(2)
```

But here, the nature of matching makes selecting the right setting tricky:

```
>>> e = x**(1 + y)
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**-a, exact=False)
x
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**-a, exact=True)
x**(-x - y + 1)
>>> (x**y).replace(x**(1 + a), lambda a: x**-a, exact=False)
x
>>> (x**y).replace(x**(1 + a), lambda a: x**-a, exact=True)
x**(1 - y)
```

It is probably better to use a different form of the query that describes the target expression more precisely:

```
>>> (1 + x**(1 + y)).replace(
... lambda x: x.is_Pow and x.exp.is_Add and x.exp.args[0] == 1,
... lambda x: x.base**(1 - (x.exp - 1)))
...
x**(1 - y) + 1
```

rewrite(**args*, *deep=True*, ***hints*)

Rewrite *self* using a defined rule.

Rewriting transforms an expression to another, which is mathematically equivalent but structurally different. For example you can rewrite trigonometric functions as complex exponentials or combinatorial functions as gamma function.

This method takes a *pattern* and a *rule* as positional arguments. *pattern* is optional parameter which defines the types of expressions that will be transformed. If it is not passed, all possible expressions will be rewritten. *rule* defines how the expression will be rewritten.

Parameters

args
[Expr] A *rule*, or *pattern* and *rule*. - *pattern* is a type or an iterable of types. - *rule* can be any object.

deep
[bool, optional] If `True`, subexpressions are recursively transformed. Default is `True`.

Examples

If *pattern* is unspecified, all possible expressions are transformed.

```
>>> from sympy import cos, sin, exp, I
>>> from sympy.abc import x
>>> expr = cos(x) + I*sin(x)
>>> expr.rewrite(exp)
exp(I*x)
```

Pattern can be a type or an iterable of types.

```
>>> expr.rewrite(sin, exp)
exp(I*x)/2 + cos(x) - exp(-I*x)/2
>>> expr.rewrite([cos,], exp)
exp(I*x)/2 + I*sin(x) + exp(-I*x)/2
>>> expr.rewrite([cos, sin], exp)
exp(I*x)
```

Rewriting behavior can be implemented by defining `_eval_rewrite()` method.

```
>>> from sympy import Expr, sqrt, pi
>>> class MySin(Expr):
...     def _eval_rewrite(self, rule, args, **hints):
...         x, = args
...         if rule == cos:
...             return cos(pi/2 - x, evaluate=False)
...         if rule == sqrt:
...             return sqrt(1 - cos(x)**2)
```

(continues on next page)

(continued from previous page)

```
>>> MySin(MySin(x)).rewrite(cos)
cos(-cos(-x + pi/2) + pi/2)
>>> MySin(x).rewrite(sqrt)
sqrt(1 - cos(x)**2)
```

Defining `_eval_rewrite_as_[...]()` method is supported for backwards compatibility reason. This may be removed in the future and using it is discouraged.

```
>>> class MySin(Expr):
...     def _eval_rewrite_as_cos(self, *args, **hints):
...         x, = args
...         return cos(pi/2 - x, evaluate=False)
>>> MySin(x).rewrite(cos)
cos(-x + pi/2)
```

round(*n=None*)

Return x rounded to the given decimal place.

If a complex number would results, apply round to the real and imaginary components of the number.

Notes

The Python `round` function uses the SymPy `round` method so it will always return a SymPy number (not a Python float or int):

```
>>> isinstance(round(S(123), -2), Number)
True
```

Examples

```
>>> from sympy import pi, E, I, S, Number
>>> pi.round()
3
>>> pi.round(2)
3.14
>>> (2*pi + E*I).round()
6 + 3*I
```

The round method has a chopping effect:

```
>>> (2*pi + I/10).round()
6
>>> (pi/10 + 2*I).round()
2*I
>>> (pi/10 + E*I).round(2)
0.31 + 2.72*I
```

separate(*deep=False*, *force=False*)

See the separate function in sympy.simplify

series(*x=None*, *x0=0*, *n=6*, *dir='+'*, *logx=None*, *cdir=0*)

Series expansion of "self" around `x = x0` yielding either terms of the series one by one (the lazy series given when n=None), else all the terms at once when n != None.

Returns the series expansion of "self" around the point `x = x0` with respect to `x` up to `O((x - x0)**n, x, x0)` (default n is 6).

If `x=None` and `self` is univariate, the univariate symbol will be supplied, otherwise an error will be raised.

Parameters

expr
[Expression] The expression whose series is to be expanded.

x
[Symbol] It is the variable of the expression to be calculated.

x0
[Value] The value around which `x` is calculated. Can be any value from `-oo` to `oo`.

n
[Value] The value used to represent the order in terms of `x**n`, up to which the series is to be expanded.

dir
[String, optional] The series-expansion can be bi-directional. If `dir="+"`, then (x->x0+). If `dir="-", then (x->x0-). For infinite ``x0` (`oo` or `-oo`), the `dir` argument is determined from the direction of the infinity (i.e., `dir="-"` for `oo`).

logx
[optional] It is used to replace any log(x) in the returned series with a symbolic value rather than evaluating the actual value.

cdir
[optional] It stands for complex direction, and indicates the direction from which the expansion needs to be evaluated.

Returns

Expr
[Expression] Series expansion of the expression about x0

Raises

TypeError
If "n" and "x0" are infinity objects

PoleError
If "x0" is an infinity object

Examples

```
>>> from sympy import cos, exp, tan
>>> from sympy.abc import x, y
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
>>> cos(x).series(n=4)
1 - x**2/2 + O(x**4)
>>> cos(x).series(x, x0=1, n=2)
```

(continues on next page)

(continued from previous page)

```
cos(1) - (x - 1)*sin(1) + O((x - 1)**2, (x, 1))
>>> e = cos(x + exp(y))
>>> e.series(y, n=2)
cos(x + 1) - y*sin(x + 1) + O(y**2)
>>> e.series(x, n=2)
cos(exp(y)) - x*sin(exp(y)) + O(x**2)
```

If n=None then a generator of the series terms will be returned.

```
>>> term=cos(x).series(n=None)
>>> [next(term) for i in range(2)]
[1, -x**2/2]
```

For dir=+ (default) the series is calculated from the right and for dir=- the series from the left. For smooth functions this flag will not alter the results.

```
>>> abs(x).series(dir="+")
x
>>> abs(x).series(dir="-")
-x
>>> f = tan(x)
>>> f.series(x, 2, 6, "+")
tan(2) + (1 + tan(2)**2)*(x - 2) + (x - 2)**2*(tan(2)**3 + tan(2)) +
(x - 2)**3*(1/3 + 4*tan(2)**2/3 + tan(2)**4) + (x - 2)**4*(tan(2)**5 +
5*tan(2)**3/3 + 2*tan(2)/3) + (x - 2)**5*(2/15 + 17*tan(2)**2/15 +
2*tan(2)**4 + tan(2)**6) + O((x - 2)**6, (x, 2))
```

```
>>> f.series(x, 2, 3, "-")
tan(2) + (2 - x)*(-tan(2)**2 - 1) + (2 - x)**2*(tan(2)**3 + tan(2))
+ O((x - 2)**3, (x, 2))
```

For rational expressions this method may return original expression without the Order term. >>> (1/x).series(x, n=8) 1/x

simplify(***kwargs*)

See the simplify function in sympy.simplify

sort_key(*order=None*)

Return a sort key.

Examples

```
>>> from sympy import S, I
```

```
>>> sorted([S(1)/2, I, -I], key=lambda x: x.sort_key())
[1/2, -I, I]
```

```
>>> S("[x, 1/x, 1/x**2, x**2, x**(1/2), x**(1/4), x**(3/2)]")
[x, 1/x, x**(-2), x**2, sqrt(x), x**(1/4), x**(3/2)]
>>> sorted(_, key=lambda x: x.sort_key())
[x**(-2), 1/x, x**(1/4), sqrt(x), x, x**(3/2), x**2]
```

subs(**args*, ***kwargs*)

Substitutes old for new in an expression after sympifying args.

***args* is either:**

- two arguments, e.g. foo.subs(old, new)
- **one iterable argument, e.g. foo.subs(iterable). The iterable may be**

  o an iterable container with (old, new) pairs. In this case the
  replacements are processed in the order given with successive patterns possibly affecting replacements already made.

  o a dict or set whose key/value items correspond to old/new pairs.
  In this case the old/new pairs will be sorted by op count and in case of a tie, by number of args and the default_sort_key. The resulting sorted list is then processed as an iterable container (see previous).

If the keyword `simultaneous` is True, the subexpressions will not be evaluated until all the substitutions have been made.

See also:

replace
replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

xreplace
exact node replacement in expr tree; also capable of using matching rules

`sympy.core.evalf.EvalfMixin.evalf`
calculates the given formula to a desired level of precision

Examples

```
>>> from sympy import pi, exp, limit, oo
>>> from sympy.abc import x, y
>>> (1 + x*y).subs(x, pi)
pi*y + 1
>>> (1 + x*y).subs({x:pi, y:2})
1 + 2*pi
>>> (1 + x*y).subs([(x, pi), (y, 2)])
1 + 2*pi
>>> reps = [(y, x**2), (x, 2)]
>>> (x + y).subs(reps)
6
>>> (x + y).subs(reversed(reps))
x**2 + 2
```

```
>>> (x**2 + x**4).subs(x**2, y)
y**2 + y
```

To replace only the x**2 but not the x**4, use xreplace:

```
>>> (x**2 + x**4).xreplace({x**2: y})
x**4 + y
```

To delay evaluation until all substitutions have been made, set the keyword `simultaneous` to True:

```
>>> (x/y).subs([(x, 0), (y, 0)])
0
>>> (x/y).subs([(x, 0), (y, 0)], simultaneous=True)
nan
```

This has the added feature of not allowing subsequent substitutions to affect those already made:

```
>>> ((x + y)/y).subs({x + y: y, y: x + y})
1
>>> ((x + y)/y).subs({x + y: y, y: x + y}, simultaneous=True)
y/(x + y)
```

In order to obtain a canonical result, unordered iterables are sorted by count_op length, number of arguments and by the default_sort_key to break any ties. All other iterables are left unsorted.

```
>>> from sympy import sqrt, sin, cos
>>> from sympy.abc import a, b, c, d, e
```

```
>>> A = (sqrt(sin(2*x)), a)
>>> B = (sin(2*x), b)
>>> C = (cos(2*x), c)
>>> D = (x, d)
>>> E = (exp(x), e)
```

```
>>> expr = sqrt(sin(2*x))*sin(exp(x)*x)*cos(2*x) + sin(2*x)
```

```
>>> expr.subs(dict([A, B, C, D, E]))
a*c*sin(d*e) + b
```

The resulting expression represents a literal replacement of the old arguments with the new arguments. This may not reflect the limiting behavior of the expression:

```
>>> (x**3 - 3*x).subs({x: oo})
nan
```

```
>>> limit(x**3 - 3*x, x, oo)
oo
```

If the substitution will be followed by numerical evaluation, it is better to pass the substitution to evalf as

```
>>> (1/x).evalf(subs={x: 3.0}, n=21)
0.333333333333333333333
```

rather than

```
>>> (1/x).subs({x: 3.0}).evalf(21)
0.333333333333333314830
```

as the former will ensure that the desired level of precision is obtained.

taylor_term(*n*, *x*, **previous_terms*)

General method for the taylor term.

This method is slow, because it differentiates n-times. Subclasses can redefine it to make it faster by using the "previous_terms".

to_nnf(*simplify=True*)

together(**args*, ***kwargs*)

See the together function in sympy.polys

transpose()

trigsimp(***args*)

See the trigsimp function in sympy.simplify

xreplace(*rule*, *hack2=False*)

Replace occurrences of objects within the expression.

Parameters

rule
[dict-like] Expresses a replacement rule

Returns

xreplace
[the result of the replacement]

See also:

replace
replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

subs
substitution of subexpressions as defined by the objects themselves.

Examples

```
>>> from sympy import symbols, pi, exp
>>> x, y, z = symbols('x y z')
>>> (1 + x*y).xreplace({x: pi})
pi*y + 1
>>> (1 + x*y).xreplace({x: pi, y: 2})
1 + 2*pi
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> (x*y + z).xreplace({x*y: pi})
z + pi
>>> (x*y*z).xreplace({x*y: pi})
x*y*z
>>> (2*x).xreplace({2*x: y, x: z})
y
>>> (2*2*x).xreplace({2*x: y, x: z})
4*z
>>> (x + y + 2).xreplace({x + y: 2})
x + y + 2
>>> (x + 2 + exp(x + 2)).xreplace({x + 2: y})
x + exp(y) + 2
```

xreplace does not differentiate between free and bound symbols. In the following, subs(x, y) would not change x since it is a bound symbol, but xreplace does:

```
>>> from sympy import Integral
>>> Integral(x, (x, 1, 2*x)).xreplace({x: y})
Integral(y, (y, 1, 2*y))
```

Trying to replace x with an expression raises an error:

```
>>> Integral(x, (x, 1, 2*x)).xreplace({x: 2*y})
ValueError: Invalid limits given: ((2*y, 1, 4*y),)
```

58.3 Functions

`nipy.algorithms.statistics.formula.formulae.`**`contrast_from_cols_or_rows`**`(`*L*, *D*, *pseudo=None*`)`

Construct a contrast matrix from a design matrix D

(possibly with its pseudo inverse already computed) and a matrix L that either specifies something in the column space of D or the row space of D.

Parameters

L
[ndarray] Matrix used to try and construct a contrast.

D
[ndarray] Design matrix used to create the contrast.

pseudo
[None or array-like, optional] If not None, gives pseudo-inverse of *D*. Allows you to pass this if it is already calculated.

Returns

C
[ndarray] Matrix with C.shape[1] == D.shape[1] representing an estimable contrast.

Notes

From an n x p design matrix D and a matrix L, tries to determine a p x q contrast matrix C which determines a contrast of full rank, i.e. the n x q matrix

dot(transpose(C), pinv(D))

is full rank.

L must satisfy either L.shape[0] == n or L.shape[1] == p.

If L.shape[0] == n, then L is thought of as representing columns in the column space of D.

If L.shape[1] == p, then L is thought of as what is known as a contrast matrix. In this case, this function returns an estimable contrast corresponding to the dot(D, L.T)

This always produces a meaningful contrast, not always with the intended properties because q is always non-zero unless L is identically 0. That is, it produces a contrast that spans the column space of L (after projection onto the column space of D).

`nipy.algorithms.statistics.formula.formulae.`**`define`**`(`**args*, ***kwargs*`)`

`nipy.algorithms.statistics.formula.formulae.`**`getparams`**`(`*expression*`)`

Return the parameters of an expression that are not Term instances but are instances of sympy.Symbol.

Examples

```
>>> x, y, z = [Term(l) for l in 'xyz']
>>> f = Formula([x,y,z])
>>> getparams(f)
[]
>>> f.mean
_b0*x + _b1*y + _b2*z
>>> getparams(f.mean)
[_b0, _b1, _b2]
>>> th = sympy.Symbol('theta')
>>> f.mean*sympy.exp(th)
(_b0*x + _b1*y + _b2*z)*exp(theta)
>>> getparams(f.mean*sympy.exp(th))
[_b0, _b1, _b2, theta]
```

nipy.algorithms.statistics.formula.formulae.**getterms**(*expression*)

Return the all instances of Term in an expression.

Examples

```
>>> x, y, z = [Term(l) for l in 'xyz']
>>> f = Formula([x,y,z])
>>> getterms(f)
[x, y, z]
>>> getterms(f.mean)
[x, y, z]
```

nipy.algorithms.statistics.formula.formulae.**is_factor**(*obj*)

Is obj a Factor?

nipy.algorithms.statistics.formula.formulae.**is_factor_term**(*obj*)

Is obj a FactorTerm?

nipy.algorithms.statistics.formula.formulae.**is_formula**(*obj*)

Is obj a Formula?

nipy.algorithms.statistics.formula.formulae.**is_term**(*obj*)

Is obj a Term?

nipy.algorithms.statistics.formula.formulae.**make_dummy**(*name*)

make_dummy is deprecated! Please use sympy.Dummy instead of this function

Make dummy variable of given name

Parameters

name
[str] name of dummy variable

Returns

dum
[*Dummy* instance]

Notes

The interface to Dummy changed between 0.6.7 and 0.7.0, and we used this function to keep compatibility. Now we depend on sympy 0.7.0 and this function is obsolete.

nipy.algorithms.statistics.formula.formulae.**make_recarray**(*rows*, *names*, *dtypes=None*, *drop_name_dim=<class 'nipy.utils._NoValue'>*)

Create recarray from *rows* with field names *names*

Create a recarray with named columns from a list or ndarray of *rows* and sequence of *names* for the columns. If *rows* is an ndarray, *dtypes* must be None, otherwise we raise a ValueError. Otherwise, if *dtypes* is None, we cast the data in all columns in *rows* as np.float64. If *dtypes* is not None, the routine uses *dtypes* as a dtype specifier for the output structured array.

Parameters

rows: list or array
Rows that will be turned into an recarray.

names: sequence
Sequence of strings - names for the columns.

dtypes: None or sequence of str or sequence of np.dtype, optional
Used to create a np.dtype, can be sequence of np.dtype or string.

drop_name_dim
[{_NoValue, False, True}, optional] Flag for compatibility with future default behavior. Current default is False. If True, drops the length 1 dimension corresponding to the axis transformed into fields when converting into a recarray. If _NoValue specified, gives default. Default will change to True in the next version of Nipy.

Returns

v
[np.ndarray] Structured array with field names given by *names*.

Raises

ValueError
dtypes not None when *rows* is array.

Examples

The following tests depend on machine byte order for their exact output.

```
>>> arr = np.array([[3, 4], [4, 6], [6, 8]])
>>> make_recarray(arr, ['x', 'y'],
...               drop_name_dim=True)
array([(3, 4), (4, 6), (6, 8)],
      dtype=[('x', '<i8'), ('y', '<i8')])
>>> make_recarray(arr, ['x', 'y'],
...               drop_name_dim=False)
array([[(3, 4)],
       [(4, 6)],
       [(6, 8)]],
      dtype=[('x', '<i8'), ('y', '<i8')])
>>> r = make_recarray(arr, ['w', 'u'], drop_name_dim=True)
```

(continues on next page)

(continued from previous page)

```
>>> make_recarray(r, ['x', 'y'],
...               drop_name_dim=True)
array([(3, 4), (4, 6), (6, 8)],
      dtype=[('x', '<i8'), ('y', '<i8')])
>>> make_recarray([[3, 4], [4, 6], [7, 9]], 'wv',
...               [np.float64, np.int_])
array([(3.0, 4), (4.0, 6), (7.0, 9)],
      dtype=[('w', '<f8'), ('v', '<i8')])
```

nipy.algorithms.statistics.formula.formulae.**natural_spline**(*t*, *knots=None*, *order=3*, *intercept=False*)

Return a Formula containing a natural spline

Spline for a Term with specified *knots* and *order*.

Parameters

t
[Term]

knots
[None or sequence, optional] Sequence of float. Default None (same as empty list)

order
[int, optional] Order of the spline. Defaults to a cubic (==3)

intercept
[bool, optional] If True, include a constant function in the natural spline. Default is False

Returns

formula
[Formula] A Formula with (len(knots) + order) Terms (if intercept=False, otherwise includes one more Term), made up of the natural spline functions.

Examples

```
>>> x = Term('x')
>>> n = natural_spline(x, knots=[1,3,4], order=3)
>>> xval = np.array([3,5,7.]).view(np.dtype([('x', np.float64)]))
>>> n.design(xval, return_float=True)
array([[   3.,    9.,   27.,    8.,    0.,   -0.],
       [   5.,   25.,  125.,   64.,    8.,    1.],
       [   7.,   49.,  343.,  216.,   64.,   27.]])
>>> d = n.design(xval)
>>> print(d.dtype.descr)
[('ns_1(x)', '<f8'), ('ns_2(x)', '<f8'), ('ns_3(x)', '<f8'), ('ns_4(x)', '<f8'), (
↪'ns_5(x)', '<f8'), ('ns_6(x)', '<f8')]
```

nipy.algorithms.statistics.formula.formulae.**terms**(*names*, ***kwargs*)

Return list of terms with names given by *names*

This is just a convenience in defining a set of terms, and is the equivalent of `sympy.symbols` for defining symbols in sympy.

We enforce the sympy 0.7.0 behavior of returning symbol "abc" from input "abc", rthan than 3 symbols "a", "b", "c".

Parameters

names
[str or sequence of str] If a single str, can specify multiple ``Term``s with string containing space or ',' as separator.

****kwargs**
[keyword arguments] keyword arguments as for `sympy.symbols`

Returns

ts
[`Term` or tuple] `Term` instance or list of `Term` instance objects named from *names*

Examples

```
>>> terms(('a', 'b', 'c'))
(a, b, c)
>>> terms('a, b, c')
(a, b, c)
>>> terms('abc')
abc
```

CHAPTER

FIFTYNINE

ALGORITHMS.STATISTICS.MIXED_EFFECTS_STAT

59.1 Module: `algorithms.statistics.mixed_effects_stat`

Inheritance diagram for `nipy.algorithms.statistics.mixed_effects_stat`:

Module for computation of mixed effects statistics with an EM algorithm. i.e. solves problems of the form y = X beta + e1 + e2, where X and Y are known, e1 and e2 are centered with diagonal covariance. V1 = var(e1) is known, and V2 = var(e2) = lambda identity. the code estimates beta and lambda using an EM algorithm. Likelihood ratio tests can then be used to test the columns of beta.

Author: Bertrand Thirion, 2012.

```
>>> N, P = 15, 500
>>> V1 = np.random.randn(N, P) ** 2
>>> effects = np.ones(P)
>>> Y = generate_data(np.ones(N), effects, .25, V1)
>>> T1 = one_sample_ttest(Y, V1, n_iter=5)
>>> T2 = t_stat(Y)
>>> assert(T1.std() < T2.std())
```

59.2 Class

59.3 `MixedEffectsModel`

class `nipy.algorithms.statistics.mixed_effects_stat.`**`MixedEffectsModel`**(*X*, *n_iter=5*, *verbose=False*)

Bases: `object`

Class to handle multiple one-sample mixed effects models

`__init__`(*X*, *n_iter=5*, *verbose=False*)

Set the effects and first-level variance, and initialize related quantities

Parameters

X: array of shape(n_samples, n_effects),
the design matrix

n_iter: int, optional,
number of iterations of the EM algorithm

verbose: bool, optional, verbosity mode

`fit`(*Y*, *V1*)
Launches the EM algorithm to estimate self

Parameters

Y, array of shape (n_samples, n_tests) or (n_samples)
the estimated effects

V1, array of shape (n_samples, n_tests) or (n_samples)
first-level variance

Returns

self

`log_like`(*Y*, *V1*)
Compute the log-likelihood of (Y, V1) under the model

Parameters

Y, array of shape (n_samples, n_tests) or (n_samples)
the estimated effects

V1, array of shape (n_samples, n_tests) or (n_samples)
first-level variance

Returns

logl: array of shape self.n_tests,
the log-likelihood of the model

`predict`(*Y*, *V1*)
Return the log_likelihood of the data.See the log_like method

`score`(*Y*, *V1*)
Return the log_likelihood of the data. See the log_like method

59.4 Functions

`nipy.algorithms.statistics.mixed_effects_stat.check_arrays`(*Y*, *V1*)
Check that the given data can be used for the models

Parameters

Y: array of shape (n_samples, n_tests) or (n_samples)
the estimated effects

V1: array of shape (n_samples, n_tests) or (n_samples)
first-level variance

`nipy.algorithms.statistics.mixed_effects_stat.`**`generate_data`**(*X*, *beta*, *V2*, *V1*)

Generate a group of individuals from the provided parameters

Parameters

X: array of shape (n_samples, n_reg),
the design matrix of the model

beta: float or array of shape (n_reg, n_tests),
the associated effects

V2: float or array of shape (n_tests),
group variance

V1: array of shape(n_samples, n_tests),
the individual variances

Returns

Y: array of shape(n_samples, n_tests)
the individual data related to the two-level normal model

`nipy.algorithms.statistics.mixed_effects_stat.`**`mfx_stat`**(*Y*, *V1*, *X*, *column*, *n_iter=5*, *return_t=True*, *return_f=False*, *return_effect=False*, *return_var=False*, *verbose=False*)

Run a mixed-effects model test on the column of the design matrix

Parameters

Y: array of shape (n_samples, n_tests)
the data

V1: array of shape (n_samples, n_tests)
first-level variance associated with the data

X: array of shape(n_samples, n_regressors)
the design matrix of the model

column: int,
index of the column of X to be tested

n_iter: int, optional,
number of iterations of the EM algorithm

return_t: bool, optional,
should one return the t test (True by default)

return_f: bool, optional,
should one return the F test (False by default)

return_effect: bool, optional,
should one return the effect estimate (False by default)

return_var: bool, optional,
should one return the variance estimate (False by default)

verbose: bool, optional, verbosity mode

Returns

(tstat, fstat, effect, var): tuple of arrays of shape (n_tests),
those required by the input return booleans

`nipy.algorithms.statistics.mixed_effects_stat.`**`one_sample_ftest`**`(`*Y*, *V1*, *n_iter=5*, *verbose=False*`)`

Returns the mixed effects F-stat for each row of the X (one sample test) This uses the Formula in Roche et al., NeuroImage 2007

Parameters

Y: array of shape (n_samples, n_tests)
the data

V1: array of shape (n_samples, n_tests)
first-level variance ssociated with the data

n_iter: int, optional,
number of iterations of the EM algorithm

verbose: bool, optional, verbosity mode

Returns

fstat, array of shape (n_tests),
statistical values obtained from the likelihood ratio test

sign, array of shape (n_tests),
sign of the mean for each test (allow for post-hoc signed tests)

`nipy.algorithms.statistics.mixed_effects_stat.`**`one_sample_ttest`**`(`*Y*, *V1*, *n_iter=5*, *verbose=False*`)`

Returns the mixed effects t-stat for each row of the X (one sample test) This uses the Formula in Roche et al., NeuroImage 2007

Parameters

Y: array of shape (n_samples, n_tests)
the observations

V1: array of shape (n_samples, n_tests)
first-level variance associated with the observations

n_iter: int, optional,
number of iterations of the EM algorithm

verbose: bool, optional, verbosity mode

Returns

tstat: array of shape (n_tests),
statistical values obtained from the likelihood ratio test

`nipy.algorithms.statistics.mixed_effects_stat.`**`t_stat`**`(`*Y*`)`

Returns the t stat of the sample on each row of the matrix

Parameters

Y, array of shape (n_samples, n_tests)

Returns

t_variates, array of shape (n_tests)

`nipy.algorithms.statistics.mixed_effects_stat.`**`two_sample_ftest`**`(`*Y*, *V1*, *group*, *n_iter=5*, *verbose=False*`)`

Returns the mixed effects t-stat for each row of the X (one sample test) This uses the Formula in Roche et al., NeuroImage 2007

Parameters

Y: array of shape (n_samples, n_tests)
the data

V1: array of shape (n_samples, n_tests)
first-level variance associated with the data

group: array of shape (n_samples)
a vector of indicators yielding the samples membership

n_iter: int, optional,
number of iterations of the EM algorithm

verbose: bool, optional, verbosity mode

Returns

tstat: array of shape (n_tests),
statistical values obtained from the likelihood ratio test

`nipy.algorithms.statistics.mixed_effects_stat.`**`two_sample_ttest`**(*Y*, *V1*, *group*, *n_iter=5*, *verbose=False*)

Returns the mixed effects t-stat for each row of the X (one sample test) This uses the Formula in Roche et al., NeuroImage 2007

Parameters

Y: array of shape (n_samples, n_tests)
the data

V1: array of shape (n_samples, n_tests)
first-level variance associated with the data

group: array of shape (n_samples)
a vector of indicators yielding the samples membership

n_iter: int, optional,
number of iterations of the EM algorithm

verbose: bool, optional, verbosity mode

Returns

tstat: array of shape (n_tests),
statistical values obtained from the likelihood ratio test

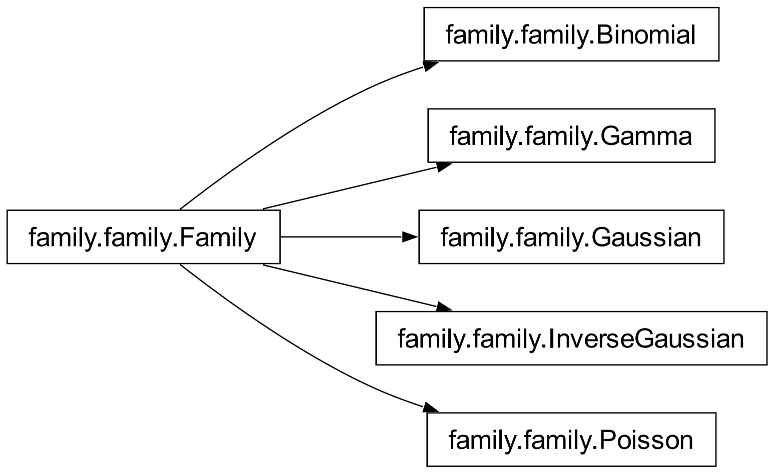
CHAPTER

SIXTY

ALGORITHMS.STATISTICS.MODELS.FAMILY.FAMILY

60.1 Module: `algorithms.statistics.models.family.family`

Inheritance diagram for `nipy.algorithms.statistics.models.family.family`:



60.2 Classes

60.2.1 `Binomial`

class `nipy.algorithms.statistics.models.family.family.`**`Binomial`**(*link=<nipy.algorithms.statistics.models.family.links.*
object>, *n=1*)

Bases: *`Family`*

Binomial exponential family.

INPUTS:
link – a Link instance n – number of trials for Binomial

`__init__`(*link=<nipy.algorithms.statistics.models.family.links.Logit object>*, *n=1*)

`deviance`(*Y*, *mu*, *scale=1.0*)

Deviance of (Y,mu) pair. Deviance is usually defined as the difference

DEV = (SUM_i -2 log Likelihood(Y_i,mu_i) + 2 log Likelihood(mu_i,mu_i)) / scale

INPUTS:
Y – response variable mu – mean parameter scale – optional scale in denominator of deviance

OUTPUTS: dev
dev – DEV, as described above

`devresid`(*Y*, *mu*)

Binomial deviance residual

INPUTS:
Y – response variable mu – mean parameter

OUTPUTS: resid
resid – deviance residuals

`fitted`(*eta*)

Fitted values based on linear predictors eta.

INPUTS:

eta – values of linear predictors, say,
X beta in a generalized linear model.

OUTPUTS: mu
mu – link.inverse(eta), mean parameter based on eta

`property link`

`links = [<nipy.algorithms.statistics.models.family.links.Logit object>, <nipy.algorithms.statistics.models.family.links.CDFLink object>, <nipy.algorithms.statistics.models.family.links.CDFLink object>, <nipy.algorithms.statistics.models.family.links.Log object>, <nipy.algorithms.statistics.models.family.links.CLogLog object>]`

`predict`(*mu*)

Linear predictors based on given mu values.

INPUTS:
mu – mean parameter of one-parameter exponential family

OUTPUTS: eta

eta – link(mu), linear predictors, based on
mean parameters mu

`tol = 1e-05`

`valid = [-inf, inf]`

`variance = <nipy.algorithms.statistics.models.family.varfuncs.Binomial object>`

`weights`(*mu*)

Weights for IRLS step.

w = 1 / (link'(mu)**2 * variance(mu))

INPUTS:
mu – mean parameter in exponential family

OUTPUTS:
w – weights used in WLS step of GLM/GAM fit

60.2.2 `Family`

class `nipy.algorithms.statistics.models.family.family.`**`Family`**(*link*, *variance*)

Bases: `object`

A class to model one-parameter exponential families.

INPUTS:
link – a Link instance variance – a variance function (models means as a function

of mean)

`__init__`(*link*, *variance*)

`deviance`(*Y*, *mu*, *scale=1.0*)

Deviance of (Y,mu) pair. Deviance is usually defined as the difference

DEV = (SUM_i -2 log Likelihood(Y_i,mu_i) + 2 log Likelihood(mu_i,mu_i)) / scale

INPUTS:
Y – response variable mu – mean parameter scale – optional scale in denominator of deviance

OUTPUTS: dev
dev – DEV, as described above

`devresid`(*Y*, *mu*)

The deviance residuals, defined as the residuals in the deviance.

Without knowing the link, they default to Pearson residuals

resid_P = (Y - mu) * sqrt(weight(mu))

INPUTS:
Y – response variable mu – mean parameter

OUTPUTS: resid
resid – deviance residuals

`fitted`(*eta*)

Fitted values based on linear predictors eta.

INPUTS:

eta – values of linear predictors, say,
X beta in a generalized linear model.

OUTPUTS: mu
mu – link.inverse(eta), mean parameter based on eta

property link

links = []

predict(*mu*)

Linear predictors based on given mu values.

INPUTS:
mu – mean parameter of one-parameter exponential family

OUTPUTS: eta

eta – link(mu), linear predictors, based on
mean parameters mu

tol = 1e-05

valid = [-inf, inf]

weights(*mu*)

Weights for IRLS step.

w = 1 / (link'(mu)**2 * variance(mu))

INPUTS:
mu – mean parameter in exponential family

OUTPUTS:
w – weights used in WLS step of GLM/GAM fit

60.2.3 Gamma

class nipy.algorithms.statistics.models.family.family.**Gamma**(*link=<nipy.algorithms.statistics.models.family.links.Pow*
object>)

Bases: *Family*

Gamma exponential family.

INPUTS:
link – a Link instance

BUGS:
no deviance residuals?

__init__(*link=<nipy.algorithms.statistics.models.family.links.Power object>*)

deviance(*Y*, *mu*, *scale=1.0*)

Deviance of (Y,mu) pair. Deviance is usually defined as the difference

DEV = (SUM_i -2 log Likelihood(Y_i,mu_i) + 2 log Likelihood(mu_i,mu_i)) / scale

INPUTS:
Y – response variable mu – mean parameter scale – optional scale in denominator of deviance

OUTPUTS: dev
dev – DEV, as described above

devresid(*Y*, *mu*)

The deviance residuals, defined as the residuals in the deviance.

Without knowing the link, they default to Pearson residuals

resid_P = (Y - mu) * sqrt(weight(mu))

INPUTS:
Y – response variable mu – mean parameter

OUTPUTS: resid
resid – deviance residuals

`fitted`(*eta*)

Fitted values based on linear predictors eta.

INPUTS:

eta – values of linear predictors, say,
X beta in a generalized linear model.

OUTPUTS: mu
mu – link.inverse(eta), mean parameter based on eta

`property link`

`links = [<nipy.algorithms.statistics.models.family.links.Log object>, <nipy.algorithms.statistics.models.family.links.Power object>, <nipy.algorithms.statistics.models.family.links.Power object>]`

`predict`(*mu*)

Linear predictors based on given mu values.

INPUTS:
mu – mean parameter of one-parameter exponential family

OUTPUTS: eta

eta – link(mu), linear predictors, based on
mean parameters mu

`tol = 1e-05`

`valid = [-inf, inf]`

`variance = <nipy.algorithms.statistics.models.family.varfuncs.Power object>`

`weights`(*mu*)

Weights for IRLS step.

w = 1 / (link'(mu)**2 * variance(mu))

INPUTS:
mu – mean parameter in exponential family

OUTPUTS:
w – weights used in WLS step of GLM/GAM fit

60.2.4 Gaussian

`class nipy.algorithms.statistics.models.family.family.Gaussian`(*link=<nipy.algorithms.statistics.models.family.links. object>*)

Bases: *Family*

Gaussian exponential family.

INPUTS:
link – a Link instance

`__init__`(*link=<nipy.algorithms.statistics.models.family.links.Power object>*)

`deviance`(*Y*, *mu*, *scale=1.0*)

Deviance of (Y,mu) pair. Deviance is usually defined as the difference

DEV = (SUM_i -2 log Likelihood(Y_i,mu_i) + 2 log Likelihood(mu_i,mu_i)) / scale

INPUTS:
Y – response variable mu – mean parameter scale – optional scale in denominator of deviance

OUTPUTS: dev
dev – DEV, as described above

`devresid`(*Y*, *mu*, *scale=1.0*)

Gaussian deviance residual

INPUTS:
Y – response variable mu – mean parameter scale – optional scale in denominator (after taking sqrt)

OUTPUTS: resid
resid – deviance residuals

`fitted`(*eta*)

Fitted values based on linear predictors eta.

INPUTS:

eta – values of linear predictors, say,
X beta in a generalized linear model.

OUTPUTS: mu
mu – link.inverse(eta), mean parameter based on eta

`property link`

`links = [<nipy.algorithms.statistics.models.family.links.Log object>, <nipy.algorithms.statistics.models.family.links.Power object>, <nipy.algorithms.statistics.models.family.links.Power object>]`

`predict`(*mu*)

Linear predictors based on given mu values.

INPUTS:
mu – mean parameter of one-parameter exponential family

OUTPUTS: eta

eta – link(mu), linear predictors, based on
mean parameters mu

`tol = 1e-05`

`valid = [-inf, inf]`

`variance = <nipy.algorithms.statistics.models.family.varfuncs.VarianceFunction object>`

`weights`(*mu*)

Weights for IRLS step.

w = 1 / (link'(mu)**2 * variance(mu))

INPUTS:
mu – mean parameter in exponential family

OUTPUTS:
w – weights used in WLS step of GLM/GAM fit

60.2.5 InverseGaussian

class nipy.algorithms.statistics.models.family.family.**InverseGaussian**(*link=<nipy.algorithms.statistics.models.fam*
object>)

Bases: *Family*

InverseGaussian exponential family.

INPUTS:
link – a Link instance n – number of trials for Binomial

__init__(*link=<nipy.algorithms.statistics.models.family.links.Power object>*)

deviance(*Y*, *mu*, *scale=1.0*)

Deviance of (Y,mu) pair. Deviance is usually defined as the difference

DEV = (SUM_i -2 log Likelihood(Y_i,mu_i) + 2 log Likelihood(mu_i,mu_i)) / scale

INPUTS:
Y – response variable mu – mean parameter scale – optional scale in denominator of deviance

OUTPUTS: dev
dev – DEV, as described above

devresid(*Y*, *mu*)

The deviance residuals, defined as the residuals in the deviance.

Without knowing the link, they default to Pearson residuals

resid_P = (Y - mu) * sqrt(weight(mu))

INPUTS:
Y – response variable mu – mean parameter

OUTPUTS: resid
resid – deviance residuals

fitted(*eta*)

Fitted values based on linear predictors eta.

INPUTS:

eta – values of linear predictors, say,
X beta in a generalized linear model.

OUTPUTS: mu
mu – link.inverse(eta), mean parameter based on eta

property link

links = [<nipy.algorithms.statistics.models.family.links.Power object>, <nipy.algorithms.statistics.models.family.links.Power object>, <nipy.algorithms.statistics.models.family.links.Power object>, <nipy.algorithms.statistics.models.family.links.Log object>]

`predict`(*mu*)

Linear predictors based on given mu values.

INPUTS:
mu – mean parameter of one-parameter exponential family

OUTPUTS: eta

eta – link(mu), linear predictors, based on
mean parameters mu

`tol = 1e-05`

`valid = [-inf, inf]`

`variance = <nipy.algorithms.statistics.models.family.varfuncs.Power object>`

`weights`(*mu*)

Weights for IRLS step.

w = 1 / (link'(mu)**2 * variance(mu))

INPUTS:
mu – mean parameter in exponential family

OUTPUTS:
w – weights used in WLS step of GLM/GAM fit

60.2.6 Poisson

`class nipy.algorithms.statistics.models.family.family.Poisson`(*link=<nipy.algorithms.statistics.models.family.links.L*
object>)

Bases: *Family*

Poisson exponential family.

INPUTS:
link – a Link instance

`__init__`(*link=<nipy.algorithms.statistics.models.family.links.Log object>*)

`deviance`(*Y*, *mu*, *scale=1.0*)

Deviance of (Y,mu) pair. Deviance is usually defined as the difference

DEV = (SUM_i -2 log Likelihood(Y_i,mu_i) + 2 log Likelihood(mu_i,mu_i)) / scale

INPUTS:
Y – response variable mu – mean parameter scale – optional scale in denominator of deviance

OUTPUTS: dev
dev – DEV, as described above

`devresid`(*Y*, *mu*)

Poisson deviance residual

INPUTS:
Y – response variable mu – mean parameter

OUTPUTS: resid
resid – deviance residuals

`fitted`(*eta*)

Fitted values based on linear predictors eta.

INPUTS:

eta – values of linear predictors, say,
X beta in a generalized linear model.

OUTPUTS: mu
mu – link.inverse(eta), mean parameter based on eta

`property link`

`links = [<nipy.algorithms.statistics.models.family.links.Log object>, <nipy.algorithms.statistics.models.family.links.Power object>, <nipy.algorithms.statistics.models.family.links.Power object>]`

`predict`(*mu*)

Linear predictors based on given mu values.

INPUTS:
mu – mean parameter of one-parameter exponential family

OUTPUTS: eta

eta – link(mu), linear predictors, based on
mean parameters mu

`tol = 1e-05`

`valid = [0, inf]`

`variance = <nipy.algorithms.statistics.models.family.varfuncs.Power object>`

`weights`(*mu*)

Weights for IRLS step.

w = 1 / (link'(mu)**2 * variance(mu))

INPUTS:
mu – mean parameter in exponential family

OUTPUTS:
w – weights used in WLS step of GLM/GAM fit

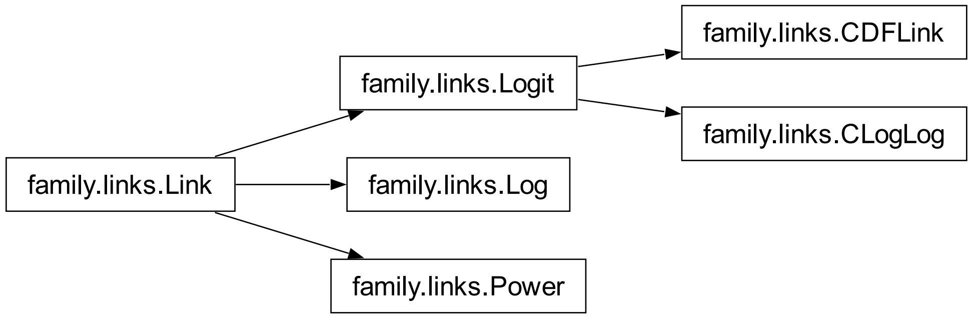
CHAPTER

SIXTYONE

ALGORITHMS.STATISTICS.MODELS.FAMILY.LINKS

61.1 Module: `algorithms.statistics.models.family.links`

Inheritance diagram for `nipy.algorithms.statistics.models.family.links`:



61.2 Classes

61.2.1 `CDFLink`

class `nipy.algorithms.statistics.models.family.links.`**`CDFLink`**(*dbn=<scipy.stats._continuous_distns.norm_gen object>*)

Bases: *`Logit`*

The use the CDF of a scipy.stats distribution as a link function:

g(x) = dbn.ppf(x)

`__init__`(*dbn=<scipy.stats._continuous_distns.norm_gen object>*)

`clean`(*p*)

Clip logistic values to range (tol, 1-tol)

INPUTS:

p – probabilities

OUTPUTS: pclip
pclip – clipped probabilities

deriv(*p*)

Derivative of CDF link

g(p) = 1/self.dbn.pdf(self.dbn.ppf(p))

INPUTS:
x – mean parameters

OUTPUTS: z
z – derivative of CDF transform of x

initialize(*Y*)

inverse(*z*)

Derivative of CDF link

g(z) = self.dbn.cdf(z)

INPUTS:
z – linear predictors in GLM

OUTPUTS: p
p – inverse of CDF link of z

tol = 1e-10

61.2.2 CLogLog

class nipy.algorithms.statistics.models.family.links.**CLogLog**

Bases: *Logit*

The complementary log-log transform as a link function:

g(x) = log(-log(x))

__init__(**args*, ***kwargs*)

clean(*p*)

Clip logistic values to range (tol, 1-tol)

INPUTS:
p – probabilities

OUTPUTS: pclip
pclip – clipped probabilities

deriv(*p*)

Derivatve of C-Log-Log transform

g(p) = - 1 / (log(p) * p)

INPUTS:
p – mean parameters

OUTPUTS: z
z – - 1 / (log(p) * p)

initialize(*Y*)

inverse(*z*)

Inverse of C-Log-Log transform

g(z) = exp(-exp(z))

INPUTS:
z – linear predictor scale

OUTPUTS: p
p – mean parameters

tol = 1e-10

61.2.3 Link

class nipy.algorithms.statistics.models.family.links.**Link**

Bases: `object`

A generic link function for one-parameter exponential family, with call, inverse and deriv methods.

__init__(**args*, ***kwargs*)

deriv(*p*)

initialize(*Y*)

inverse(*z*)

61.2.4 Log

class nipy.algorithms.statistics.models.family.links.**Log**

Bases: *Link*

The log transform as a link function:

g(x) = log(x)

__init__(**args*, ***kwargs*)

clean(*x*)

deriv(*x*)

Derivative of log transform

g(x) = 1/x

INPUTS:
x – mean parameters

OUTPUTS: z
z – derivative of log transform of x

initialize(*Y*)

inverse(*z*)

Inverse of log transform

g(x) = exp(x)

INPUTS:
z – linear predictors in GLM

OUTPUTS: x
x – exp(z)

tol = 1e-10

61.2.5 Logit

class nipy.algorithms.statistics.models.family.links.**Logit**

Bases: *Link*

The logit transform as a link function:

g'(x) = 1 / (x * (1 - x)) g^(-1)(x) = exp(x)/(1 + exp(x))

__init__(**args*, ***kwargs*)

clean(*p*)

Clip logistic values to range (tol, 1-tol)

INPUTS:
p – probabilities

OUTPUTS: pclip
pclip – clipped probabilities

deriv(*p*)

Derivative of logit transform

g(p) = 1 / (p * (1 - p))

INPUTS:
p – probabilities

OUTPUTS: y
y – derivative of logit transform of p

initialize(*Y*)

inverse(*z*)

Inverse logit transform

h(z) = exp(z)/(1+exp(z))

INPUTS:
z – logit transform of p

OUTPUTS: p
p – probabilities

tol = 1e-10

61.2.6 Power

class nipy.algorithms.statistics.models.family.links.**Power**(*power=1.0*)

Bases: *Link*

The power transform as a link function:

g(x) = x**power

__init__(*power=1.0*)

deriv(*x*)

Derivative of power transform

g(x) = self.power * x**(self.power - 1)

INPUTS:
x – mean parameters

OUTPUTS: z
z – derivative of power transform of x

initialize(*Y*)

inverse(*z*)

Inverse of power transform

g(x) = x**(1/self.power)

INPUTS:
z – linear predictors in GLM

OUTPUTS: x
x – mean parameters

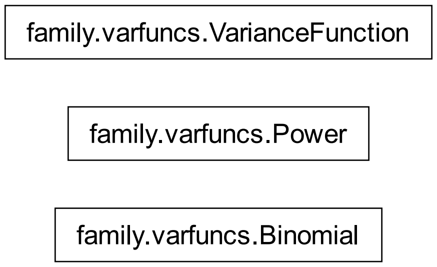
CHAPTER

SIXTYTWO

ALGORITHMS.STATISTICS.MODELS.FAMILY.VARFUNCS

62.1 Module: `algorithms.statistics.models.family.varfuncs`

Inheritance diagram for `nipy.algorithms.statistics.models.family.varfuncs`:



62.2 Classes

62.2.1 `Binomial`

class `nipy.algorithms.statistics.models.family.varfuncs.`**`Binomial`**(*n=1*)

Bases: `object`

Binomial variance function

p = mu / n; V(mu) = p * (1 - p) * n

INPUTS:
n – number of trials in Binomial

`__init__`(*n=1*)

`clean`(*p*)

`tol = 1e-10`

62.2.2 `Power`

class `nipy.algorithms.statistics.models.family.varfuncs.`**`Power`**(*power=1.0*)

Bases: `object`

Power variance function:

V(mu) = fabs(mu)**power

INPUTS:
power – exponent used in power variance function

`__init__`(*power=1.0*)

62.2.3 `VarianceFunction`

class `nipy.algorithms.statistics.models.family.varfuncs.`**`VarianceFunction`**

Bases: `object`

Variance function that relates the variance of a random variable to its mean. Defaults to 1.

`__init__`(**args*, ***kwargs*)

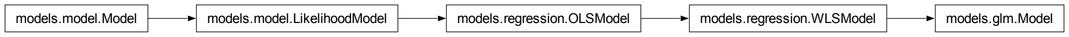
CHAPTER

SIXTYTHREE

ALGORITHMS.STATISTICS.MODELS.GLM

63.1 Module: `algorithms.statistics.models.glm`

Inheritance diagram for `nipy.algorithms.statistics.models.glm`:


models.model.Model
models.model.LikelihoodModel
models.regression.OLSModel
models.regression.WLSModel
models.glm.Model

63.1.1 General linear models

63.2 `Model`

class `nipy.algorithms.statistics.models.glm.`**`Model`**(*design*, *family=<nipy.algorithms.statistics.models.family.family.Gaussian object>*)

Bases: *`WLSModel`*

`__init__`(*design*, *family=<nipy.algorithms.statistics.models.family.family.Gaussian object>*)

Parameters

design
[array-like] This is your design matrix. Data are assumed to be column ordered with observations in rows.

`cont`(*tol=1e-05*)

Continue iterating, or has convergence been obtained?

`deviance`(*Y=None*, *results=None*, *scale=1.0*)

Return (unnormalized) log-likelihood for GLM.

Note that self.scale is interpreted as a variance in old_model, so we divide the residuals by its sqrt.

`estimate_scale`(*Y=None*, *results=None*)

Return Pearson's X^2 estimate of scale.

`fit`(*Y*)

Fit model to data *Y*

Full fit of the model including estimate of covariance matrix, (whitened) residuals and scale.

Parameters

Y
[array-like] The dependent variable for the Least Squares problem.

Returns

fit
[RegressionResults]

`has_intercept`()

Check if column of 1s is in column space of design

`information`(*beta*, *nuisance=None*)

Returns the information matrix at (beta, Y, nuisance).

See logL for details.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

nuisance
[dict] A dict with key 'sigma', which is an estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n` where n=Y.shape[0], X=self.design.

Returns

info
[array] The information matrix, the negative of the inverse of the Hessian of the of the log-likelihood function evaluated at (theta, Y, nuisance).

`initialize`(*design*)

Initialize (possibly re-initialize) a Model instance.

For instance, the design matrix of a linear model may change and some things must be recomputed.

`logL`(*beta*, *Y*, *nuisance=None*)

Returns the value of the loglikelihood function at beta.

Given the whitened design matrix, the loglikelihood is evaluated at the parameter vector, beta, for the dependent variable, Y and the nuisance parameter, sigma.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

Y
[ndarray] The dependent variable

nuisance
[dict, optional] A dict with key 'sigma', which is an optional estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n`, where n=Y.shape[0], X=self.design.

Returns

loglf
[float] The value of the loglikelihood function.

Notes

The log-Likelihood Function is defined as

$$\ell(\beta,\sigma,Y) = -\frac{n}{2}\log(2\pi\sigma^2) - \|Y-X\beta\|^2/(2\sigma^2)$$

The parameter σ above is what is sometimes referred to as a nuisance parameter. That is, the likelihood is considered as a function of β, but to evaluate it, a value of σ is needed.

If σ is not provided, then its maximum likelihood estimate:

$$\hat{\sigma}(\beta) = \frac{\text{SSE}(\beta)}{n}$$

is plugged in. This likelihood is now a function of only β and is technically referred to as a profile-likelihood.

References

[1]

niter = 10

predict(*design=None*)

After a model has been fit, results are (assumed to be) stored in self.results, which itself should have a predict method.

rank()

Compute rank of design matrix

score(*beta*, *Y*, *nuisance=None*)

Gradient of the loglikelihood function at (beta, Y, nuisance).

The graient of the loglikelihood function at (beta, Y, nuisance) is the score function.

See `logL()` for details.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

Y
[ndarray] The dependent variable.

nuisance
[dict, optional] A dict with key 'sigma', which is an optional estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n`, where n=Y.shape[0], X=self.design.

Returns

The gradient of the loglikelihood function.

whiten(*X*)

Whitener for WLS model, multiplies by sqrt(self.weights)

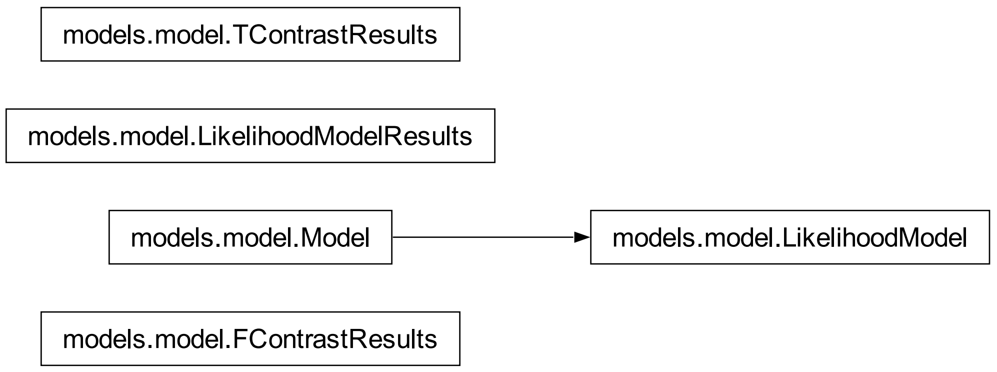
CHAPTER

SIXTYFOUR

ALGORITHMS.STATISTICS.MODELS.MODEL

64.1 Module: `algorithms.statistics.models.model`

Inheritance diagram for `nipy.algorithms.statistics.models.model`:


models.model.TContrastResults
models.model.LikelihoodModelResults
models.model.Model
models.model.LikelihoodModel
models.model.FContrastResults

64.2 Classes

64.2.1 `FContrastResults`

class `nipy.algorithms.statistics.models.model.`**`FContrastResults`**(*effect*, *covariance*, *F*, *df_num*, *df_den=None*)

Bases: `object`

Results from an F contrast of coefficients in a parametric model.

The class does nothing, it is a container for the results from F contrasts, and returns the F-statistics when np.asarray is called.

`__init__`(*effect*, *covariance*, *F*, *df_num*, *df_den=None*)

64.2.2 `LikelihoodModel`

class nipy.algorithms.statistics.models.model.**LikelihoodModel**

Bases: *Model*

`__init__`()

fit()

Fit a model to data.

information(*theta*, *nuisance=None*)

Fisher information matrix

The inverse of the expected value of `- d^2 logL / dtheta^2.`

initialize()

Initialize (possibly re-initialize) a Model instance.

For instance, the design matrix of a linear model may change and some things must be recomputed.

logL(*theta*, *Y*, *nuisance=None*)

Log-likelihood of model.

predict(*design=None*)

After a model has been fit, results are (assumed to be) stored in self.results, which itself should have a predict method.

score(*theta*, *Y*, *nuisance=None*)

Gradient of logL with respect to theta.

This is the score function of the model

64.2.3 `LikelihoodModelResults`

class nipy.algorithms.statistics.models.model.**LikelihoodModelResults**(*theta*, *Y*, *model*, *cov=None*, *dispersion=1.0*, *nuisance=None*, *rank=None*)

Bases: `object`

Class to contain results from likelihood models

`__init__`(*theta*, *Y*, *model*, *cov=None*, *dispersion=1.0*, *nuisance=None*, *rank=None*)

Set up results structure

Parameters

theta
[ndarray] parameter estimates from estimated model

Y
[ndarray] data

model
[`LikelihoodModel` instance] model used to generate fit

cov
[None or ndarray, optional] covariance of thetas

dispersion
[scalar, optional] multiplicative factor in front of *cov*

nuisance
[None of ndarray] parameter estimates needed to compute logL

rank
[None or scalar] rank of the model. If rank is not None, it is used for df_model instead of the usual counting of parameters.

Notes

The covariance of thetas is given by:

dispersion * cov

For (some subset of models) *dispersion* will typically be the mean square error from the estimated model (sigma^2)

`AIC()`

Akaike Information Criterion

`BIC()`

Schwarz's Bayesian Information Criterion

`Fcontrast`(*matrix*, *dispersion=None*, *invcov=None*)

Compute an Fcontrast for a contrast matrix *matrix*.

Here, *matrix* M is assumed to be non-singular. More precisely

$$MpXpX'M'$$

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

See the contrast module to see how to specify contrasts. In particular, the matrices from these contrasts will always be non-singular in the sense above.

Parameters

matrix
[1D array-like] contrast matrix

dispersion
[None or float, optional] If None, use `self.dispersion`

invcov
[None or array, optional] Known inverse of variance covariance matrix. If None, calculate this matrix.

Returns

f_res
[`FContrastResults` instance] with attributes F, df_den, df_num

Notes

For F contrasts, we now specify an effect and covariance

`Tcontrast`(*matrix*, *store=('t', 'effect', 'sd')*, *dispersion=None*)

Compute a Tcontrast for a row vector *matrix*

To get the t-statistic for a single column, use the 't' method.

Parameters

matrix
[1D array-like] contrast matrix

store
[sequence, optional] components of t to store in results output object. Defaults to all components ('t', 'effect', 'sd').

dispersion
[None or float, optional]

Returns

res
[`TContrastResults` object]

`conf_int`(*alpha=0.05*, *cols=None*, *dispersion=None*)

The confidence interval of the specified theta estimates.

Parameters

alpha
[float, optional] The *alpha* level for the confidence interval. ie., *alpha* = .05 returns a 95% confidence interval.

cols
[tuple, optional] *cols* specifies which confidence intervals to return

dispersion
[None or scalar] scale factor for the variance / covariance (see class docstring and `vcov` method docstring)

Returns

cis
[ndarray] *cis* is shape `(len(cols), 2)` where each row contains [lower, upper] for the given entry in *cols*

Notes

Confidence intervals are two-tailed. TODO: tails : string, optional

tails can be "two", "upper", or "lower"

Examples

```
>>> from numpy.random import standard_normal as stan
>>> from nipy.algorithms.statistics.models.regression import OLSModel
>>> x = np.hstack((stan((30,1)),stan((30,1)),stan((30,1))))
>>> beta=np.array([3.25, 1.5, 7.0])
>>> y = np.dot(x,beta) + stan((30))
>>> model = OLSModel(x).fit(y)
>>> confidence_intervals = model.conf_int(cols=(1,2))
```

`logL()`

The maximized log-likelihood

`t`(*column=None*)

Return the (Wald) t-statistic for a given parameter estimate.

Use Tcontrast for more complicated (Wald) t-statistics.

`vcov`(*matrix=None*, *column=None*, *dispersion=None*, *other=None*)

Variance/covariance matrix of linear contrast

Parameters

matrix: (dim, self.theta.shape[0]) array, optional
numerical contrast specification, where `dim` refers to the 'dimension' of the contrast i.e. 1 for t contrasts, 1 or more for F contrasts.

column: int, optional
alternative way of specifying contrasts (column index)

dispersion: float or (n_voxels,) array, optional
value(s) for the dispersion parameters

other: (dim, self.theta.shape[0]) array, optional
alternative contrast specification (?)

Returns

cov: (dim, dim) or (n_voxels, dim, dim) array
the estimated covariance matrix/matrices

Returns the variance/covariance matrix of a linear contrast of the estimates of theta, multiplied by *dispersion* which will often be an estimate of *dispersion*, like, sigma^2.
The covariance of interest is either specified as a (set of) column(s) or a matrix.

64.2.4 `Model`

class `nipy.algorithms.statistics.models.model.`**`Model`**

Bases: `object`

A (predictive) statistical model.

The class Model itself does nothing but lays out the methods expected of any subclass.

`__init__``()`

fit()

Fit a model to data.

initialize()

Initialize (possibly re-initialize) a Model instance.

For instance, the design matrix of a linear model may change and some things must be recomputed.

predict(*design=None*)

After a model has been fit, results are (assumed to be) stored in self.results, which itself should have a predict method.

64.2.5 TContrastResults

class nipy.algorithms.statistics.models.model.**TContrastResults**(*t*, *sd*, *effect*, *df_den=None*)

Bases: `object`

Results from a t contrast of coefficients in a parametric model.

The class does nothing, it is a container for the results from T contrasts, and returns the T-statistics when np.asarray is called.

__init__(*t*, *sd*, *effect*, *df_den=None*)

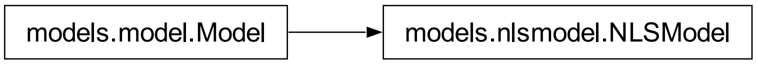
CHAPTER

SIXTYFIVE

ALGORITHMS.STATISTICS.MODELS.NLSMODEL

65.1 Module: `algorithms.statistics.models.nlsmodel`

Inheritance diagram for `nipy.algorithms.statistics.models.nlsmodel`:



Non-linear least squares model

65.2 `NLSModel`

class `nipy.algorithms.statistics.models.nlsmodel.`**`NLSModel`**(*Y*, *design*, *f*, *grad*, *theta*, *niter=10*)

Bases: `Model`

Class representing a simple nonlinear least squares model.

`__init__`(*Y*, *design*, *f*, *grad*, *theta*, *niter=10*)

Initialize non-linear model instance

Parameters

Y
[ndarray] the data in the NLS model

design
[ndarray] the design matrix, X

f
[callable] the map between the (linear parameters (in the design matrix) and the nonlinear parameters (theta)) and the predicted data. f accepts the design matrix and the parameters (theta) as input, and returns the predicted data at that design.

grad
[callable] 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.

theta
[array] parameters

niter
[int] number of iterations

`SSE()`

Sum of squares error.

Returns

sse: float
sum of squared residuals

`fit()`

Fit a model to data.

`getZ()`

Set Z into *self*

Returns

None

`getomega()`

Set omega into *self*

Returns

None

`initialize()`

Initialize (possibly re-initialize) a Model instance.

For instance, the design matrix of a linear model may change and some things must be recomputed.

`predict`(*design=None*)

Get predicted values for *design* or `self.design`

Parameters

design
[None or array, optional] design at which to predict data. If None (the default) then use the initial `self.design`

Returns

y_predicted
[array] predicted data at given (or initial) design

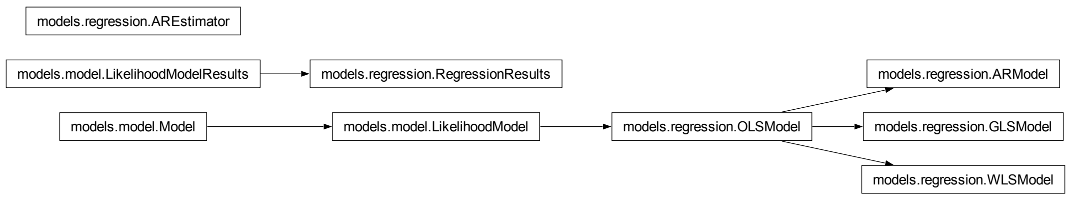
CHAPTER

SIXTYSIX

ALGORITHMS.STATISTICS.MODELS.REGRESSION

66.1 Module: `algorithms.statistics.models.regression`

Inheritance diagram for `nipy.algorithms.statistics.models.regression`:


models.regression.AREstimator
models.model.LikelihoodModelResults
models.regression.RegressionResults
models.regression.ARModel
models.model.Model
models.model.LikelihoodModel
models.regression.OLSModel
models.regression.GLSModel
models.regression.WLSModel

This module implements some standard regression models: OLS and WLS models, as well as an AR(p) regression model.

Models are specified with a design matrix and are fit using their 'fit' method.

Subclasses that have more complicated covariance matrices should write over the 'whiten' method as the fit method prewhitens the response by calling 'whiten'.

General reference for regression models:

'Introduction to Linear Regression Analysis', Douglas C. Montgomery,
Elizabeth A. Peck, G. Geoffrey Vining. Wiley, 2006.

66.2 Classes

66.2.1 `AREstimator`

class `nipy.algorithms.statistics.models.regression.`**`AREstimator`**(*model*, *p=1*)

Bases: `object`

A class to estimate AR(p) coefficients from residuals

`__init__`(*model*, *p=1*)

Bias-correcting AR estimation class

Parameters

model
[OSLModel instance] A models.regression.OLSmodel instance, where *model* has attribute design

p
[int, optional] Order of AR(p) noise

66.2.2 ARModel

class nipy.algorithms.statistics.models.regression.**ARModel**(*design*, *rho*)

Bases: *OLSModel*

A regression model with an AR(p) covariance structure.

In terms of a LikelihoodModel, the parameters are beta, the usual regression parameters, and sigma, a scalar nuisance parameter that shows up as multiplier in front of the AR(p) covariance.

The linear autoregressive process of order p–AR(p)–is defined as:
TODO

Examples

```
>>> from nipy.algorithms.statistics.api import Term, Formula
>>> data = np.rec.fromarrays(([1,3,4,5,8,10,9], range(1,8)),
...                          names=('Y', 'X'))
>>> f = Formula([Term("X"), 1])
>>> dmtx = f.design(data, return_float=True)
>>> model = ARModel(dmtx, 2)
```

We go through the model.iterative_fit procedure long-hand:

```
>>> for i in range(6):
...     results = model.fit(data['Y'])
...     print("AR coefficients:", model.rho)
...     rho, sigma = yule_walker(data["Y"] - results.predicted,
...                              order=2,
...                              df=model.df_resid)
...     model = ARModel(model.design, rho)
...
AR coefficients: [ 0.  0.]
AR coefficients: [-0.61530877 -1.01542645]
AR coefficients: [-0.72660832 -1.06201457]
AR coefficients: [-0.7220361  -1.05365352]
AR coefficients: [-0.72229201 -1.05408193]
AR coefficients: [-0.722278   -1.05405838]
>>> results.theta
array([ 1.59564228, -0.58562172])
>>> results.t()
array([ 38.0890515 ,  -3.45429252])
>>> print(results.Tcontrast([0,1]))
<T contrast: effect=-0.58562172384377043, sd=0.16953449108110835,
t=-3.4542925165805847, df_den=5>
>>> print(results.Fcontrast(np.identity(2)))
<F contrast: F=4216.810299725842, df_den=5, df_num=2>
```

Reinitialize the model, and do the automated iterative fit

```
>>> model.rho = np.array([0,0])
>>> model.iterative_fit(data['Y'], niter=3)
>>> print(model.rho)
[-0.7220361  -1.05365352]
```

`__init__`(*design*, *rho*)

Initialize AR model instance

Parameters

design
[ndarray] 2D array with design matrix

rho
[int or array-like] If int, gives order of model, and initializes rho to zeros. If ndarray, gives initial estimate of rho. Be careful as `ARModel(X, 1) != ARModel(X, 1.0)`.

`fit`(*Y*)

Fit model to data *Y*

Full fit of the model including estimate of covariance matrix, (whitened) residuals and scale.

Parameters

Y
[array-like] The dependent variable for the Least Squares problem.

Returns

fit
[RegressionResults]

`has_intercept`()

Check if column of 1s is in column space of design

`information`(*beta*, *nuisance=None*)

Returns the information matrix at (beta, Y, nuisance).

See logL for details.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

nuisance
[dict] A dict with key 'sigma', which is an estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n` where n=Y.shape[0], X=self.design.

Returns

info
[array] The information matrix, the negative of the inverse of the Hessian of the of the log-likelihood function evaluated at (theta, Y, nuisance).

`initialize`(*design*)

Initialize (possibly re-initialize) a Model instance.

For instance, the design matrix of a linear model may change and some things must be recomputed.

`iterative_fit`(*Y*, *niter=3*)

Perform an iterative two-stage procedure to estimate AR(p) parameters and regression coefficients simultaneously.

Parameters

Y
[ndarray] data to which to fit model

niter
[optional, int] the number of iterations (default 3)

Returns

None

`logL`(*beta*, *Y*, *nuisance=None*)

Returns the value of the loglikelihood function at beta.

Given the whitened design matrix, the loglikelihood is evaluated at the parameter vector, beta, for the dependent variable, Y and the nuisance parameter, sigma.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

Y
[ndarray] The dependent variable

nuisance
[dict, optional] A dict with key 'sigma', which is an optional estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n`, where n=Y.shape[0], X=self.design.

Returns

loglf
[float] The value of the loglikelihood function.

Notes

The log-Likelihood Function is defined as

$$\ell(\beta,\sigma,Y) = -\frac{n}{2}\log(2\pi\sigma^2) - \|Y-X\beta\|^2/(2\sigma^2)$$

The parameter σ above is what is sometimes referred to as a nuisance parameter. That is, the likelihood is considered as a function of β, but to evaluate it, a value of σ is needed.

If σ is not provided, then its maximum likelihood estimate:

$$\hat{\sigma}(\beta) = \frac{\text{SSE}(\beta)}{n}$$

is plugged in. This likelihood is now a function of only β and is technically referred to as a profile-likelihood.

References

[1]

`predict`(*design=None*)

After a model has been fit, results are (assumed to be) stored in self.results, which itself should have a predict method.

`rank`()

Compute rank of design matrix

`score`(*beta*, *Y*, *nuisance=None*)

Gradient of the loglikelihood function at (beta, Y, nuisance).

The graient of the loglikelihood function at (beta, Y, nuisance) is the score function.

See *`logL()`* for details.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

Y
[ndarray] The dependent variable.

nuisance
[dict, optional] A dict with key 'sigma', which is an optional estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n`, where n=Y.shape[0], X=self.design.

Returns

The gradient of the loglikelihood function.

`whiten`(*X*)

Whiten a series of columns according to AR(p) covariance structure

Parameters

X
[array-like of shape (n_features)] array to whiten

Returns

wX
[ndarray] X whitened with order self.order AR

66.2.3 GLSModel

class `nipy.algorithms.statistics.models.regression.GLSModel`(*design*, *sigma*)

Bases: *`OLSModel`*

Generalized least squares model with a general covariance structure

`__init__`(*design*, *sigma*)

Parameters

design
[array-like] This is your design matrix. Data are assumed to be column ordered with observations in rows.

`fit`(*Y*)

Fit model to data *Y*

Full fit of the model including estimate of covariance matrix, (whitened) residuals and scale.

Parameters

Y
[array-like] The dependent variable for the Least Squares problem.

Returns

fit
[RegressionResults]

`has_intercept`()

Check if column of 1s is in column space of design

`information`(*beta*, *nuisance=None*)

Returns the information matrix at (beta, Y, nuisance).

See logL for details.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

nuisance
[dict] A dict with key 'sigma', which is an estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n` where n=Y.shape[0], X=self.design.

Returns

info
[array] The information matrix, the negative of the inverse of the Hessian of the of the log-likelihood function evaluated at (theta, Y, nuisance).

`initialize`(*design*)

Initialize (possibly re-initialize) a Model instance.

For instance, the design matrix of a linear model may change and some things must be recomputed.

`logL`(*beta*, *Y*, *nuisance=None*)

Returns the value of the loglikelihood function at beta.

Given the whitened design matrix, the loglikelihood is evaluated at the parameter vector, beta, for the dependent variable, Y and the nuisance parameter, sigma.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

Y
[ndarray] The dependent variable

nuisance
[dict, optional] A dict with key 'sigma', which is an optional estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n`, where n=Y.shape[0], X=self.design.

Returns

loglf
[float] The value of the loglikelihood function.

Notes

The log-Likelihood Function is defined as

$$\ell(\beta,\sigma,Y) = -\frac{n}{2}\log(2\pi\sigma^2) - \|Y-X\beta\|^2/(2\sigma^2)$$

The parameter σ above is what is sometimes referred to as a nuisance parameter. That is, the likelihood is considered as a function of β, but to evaluate it, a value of σ is needed.

If σ is not provided, then its maximum likelihood estimate:

$$\hat{\sigma}(\beta) = \frac{\text{SSE}(\beta)}{n}$$

is plugged in. This likelihood is now a function of only β and is technically referred to as a profile-likelihood.

References

[1]

`predict`(*design=None*)

After a model has been fit, results are (assumed to be) stored in self.results, which itself should have a predict method.

`rank`()

Compute rank of design matrix

`score`(*beta*, *Y*, *nuisance=None*)

Gradient of the loglikelihood function at (beta, Y, nuisance).

The graient of the loglikelihood function at (beta, Y, nuisance) is the score function.

See `logL()` for details.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

Y
[ndarray] The dependent variable.

nuisance
[dict, optional] A dict with key 'sigma', which is an optional estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n`, where n=Y.shape[0], X=self.design.

Returns

The gradient of the loglikelihood function.

`whiten`(*Y*)

Whiten design matrix

Parameters

X
[array] design matrix

Returns

wX
[array] This matrix is the matrix whose pseudoinverse is ultimately used in estimating the coefficients. For OLSModel, it is does nothing. For WLSmodel, ARmodel, it pre-applies a square root of the covariance matrix to X.

66.2.4 OLSModel

class `nipy.algorithms.statistics.models.regression.`**`OLSModel`**(*design*)

Bases: *LikelihoodModel*

A simple ordinary least squares model.

Parameters

design
[array-like] This is your design matrix. Data are assumed to be column ordered with observations in rows.

Examples

```
>>> from nipy.algorithms.statistics.api import Term, Formula
>>> data = np.rec.fromarrays(([1,3,4,5,2,3,4], range(1,8)),
...                          names=('Y', 'X'))
>>> f = Formula([Term("X"), 1])
>>> dmtx = f.design(data, return_float=True)
>>> model = OLSModel(dmtx)
>>> results = model.fit(data['Y'])
>>> results.theta
array([ 0.25      ,  2.14285714])
>>> results.t()
array([ 0.98019606,  1.87867287])
>>> print(results.Tcontrast([0,1]))
<T contrast: effect=2.14285714286, sd=1.14062281591, t=1.87867287326,
df_den=5>
>>> print(results.Fcontrast(np.eye(2)))
<F contrast: F=19.4607843137, df_den=5, df_num=2>
```

Attributes

design
[ndarray] This is the design, or X, matrix.

wdesign
[ndarray] This is the whitened design matrix. *design* == *wdesign* by default for the OLSModel, though models that inherit from the OLSModel will whiten the design.

calc_beta
[ndarray] This is the Moore-Penrose pseudoinverse of the whitened design matrix.

normalized_cov_beta
[ndarray] `np.dot(calc_beta, calc_beta.T)`

df_resid
[scalar] Degrees of freedom of the residuals. Number of observations less the rank of the design.

df_model
[scalar] Degrees of freedome of the model. The rank of the design.

Methods

model.__init___(design)
model.logL(b=self.beta, Y)

`__init__`(*design*)

Parameters

design
[array-like] This is your design matrix. Data are assumed to be column ordered with observations in rows.

`fit`(*Y*)

Fit model to data *Y*

Full fit of the model including estimate of covariance matrix, (whitened) residuals and scale.

Parameters

Y
[array-like] The dependent variable for the Least Squares problem.

Returns

fit
[RegressionResults]

`has_intercept`()

Check if column of 1s is in column space of design

`information`(*beta*, *nuisance=None*)

Returns the information matrix at (beta, Y, nuisance).

See logL for details.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

nuisance
[dict] A dict with key 'sigma', which is an estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n` where n=Y.shape[0], X=self.design.

Returns

info
[array] The information matrix, the negative of the inverse of the Hessian of the of the log-likelihood function evaluated at (theta, Y, nuisance).

initialize(*design*)

Initialize (possibly re-initialize) a Model instance.

For instance, the design matrix of a linear model may change and some things must be recomputed.

logL(*beta*, *Y*, *nuisance=None*)

Returns the value of the loglikelihood function at beta.

Given the whitened design matrix, the loglikelihood is evaluated at the parameter vector, beta, for the dependent variable, Y and the nuisance parameter, sigma.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

Y
[ndarray] The dependent variable

nuisance
[dict, optional] A dict with key 'sigma', which is an optional estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n`, where n=Y.shape[0], X=self.design.

Returns

loglf
[float] The value of the loglikelihood function.

Notes

The log-Likelihood Function is defined as

$$\ell(\beta,\sigma,Y)=-\frac{n}{2}\log(2\pi\sigma^2)-\|Y-X\beta\|^2/(2\sigma^2)$$

The parameter σ above is what is sometimes referred to as a nuisance parameter. That is, the likelihood is considered as a function of β, but to evaluate it, a value of σ is needed.

If σ is not provided, then its maximum likelihood estimate:

$$\hat{\sigma}(\beta)=\frac{\text{SSE}(\beta)}{n}$$

is plugged in. This likelihood is now a function of only β and is technically referred to as a profile-likelihood.

References

[1]

predict(*design=None*)

After a model has been fit, results are (assumed to be) stored in self.results, which itself should have a predict method.

rank()

Compute rank of design matrix

`score`(*beta*, *Y*, *nuisance=None*)

Gradient of the loglikelihood function at (beta, Y, nuisance).

The graient of the loglikelihood function at (beta, Y, nuisance) is the score function.

See `logL()` for details.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

Y
[ndarray] The dependent variable.

nuisance
[dict, optional] A dict with key 'sigma', which is an optional estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n`, where n=Y.shape[0], X=self.design.

Returns

The gradient of the loglikelihood function.

`whiten`(*X*)

Whiten design matrix

Parameters

X
[array] design matrix

Returns

wX
[array] This matrix is the matrix whose pseudoinverse is ultimately used in estimating the coefficients. For OLSModel, it is does nothing. For WLSmodel, ARmodel, it pre-applies a square root of the covariance matrix to X.

66.2.5 RegressionResults

`class nipy.algorithms.statistics.models.regression.RegressionResults`(*theta*, *Y*, *model*, *wY*, *wresid*, *cov=None*, *dispersion=1.0*, *nuisance=None*)

Bases: `LikelihoodModelResults`

This class summarizes the fit of a linear regression model.

It handles the output of contrasts, estimates of covariance, etc.

`__init__`(*theta*, *Y*, *model*, *wY*, *wresid*, *cov=None*, *dispersion=1.0*, *nuisance=None*)

See LikelihoodModelResults constructor.

The only difference is that the whitened Y and residual values are stored for a regression model.

`AIC`()

Akaike Information Criterion

`BIC()`

Schwarz's Bayesian Information Criterion

`F_overall()`

Overall goodness of fit F test, comparing model to a model with just an intercept. If not an OLS model this is a pseudo-F.

`Fcontrast`(*matrix*, *dispersion=None*, *invcov=None*)

Compute an Fcontrast for a contrast matrix *matrix*.

Here, *matrix* M is assumed to be non-singular. More precisely

$$MpXpX'M'$$

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

See the contrast module to see how to specify contrasts. In particular, the matrices from these contrasts will always be non-singular in the sense above.

Parameters

matrix
[1D array-like] contrast matrix

dispersion
[None or float, optional] If None, use `self.dispersion`

invcov
[None or array, optional] Known inverse of variance covariance matrix. If None, calculate this matrix.

Returns

f_res
[`FContrastResults` instance] with attributes F, df_den, df_num

Notes

For F contrasts, we now specify an effect and covariance

`MSE()`

Mean square (error)

`MSR()`

Mean square (regression)

`MST()`

Mean square (total)

`R2()`

Return the adjusted R^2 value for each row of the response Y.

Notes

Changed to the textbook definition of R^2.

See: Davidson and MacKinnon p 74

`R2_adj()`

Return the R^2 value for each row of the response Y.

Notes

Changed to the textbook definition of R^2.

See: Davidson and MacKinnon p 74

`SSE()`

Error sum of squares. If not from an OLS model this is "pseudo"-SSE.

`SSR()`

Regression sum of squares

`SST()`

Total sum of squares. If not from an OLS model this is "pseudo"-SST.

`Tcontrast`(*matrix*, *store=('t', 'effect', 'sd')*, *dispersion=None*)

Compute a Tcontrast for a row vector *matrix*

To get the t-statistic for a single column, use the 't' method.

Parameters

matrix
[1D array-like] contrast matrix

store
[sequence, optional] components of t to store in results output object. Defaults to all components ('t', 'effect', 'sd').

dispersion
[None or float, optional]

Returns

res
[`TContrastResults` object]

`conf_int`(*alpha=0.05*, *cols=None*, *dispersion=None*)

The confidence interval of the specified theta estimates.

Parameters

alpha
[float, optional] The *alpha* level for the confidence interval. ie., *alpha* = .05 returns a 95% confidence interval.

cols
[tuple, optional] *cols* specifies which confidence intervals to return

dispersion
[None or scalar] scale factor for the variance / covariance (see class docstring and `vcov` method docstring)

Returns

cis
[ndarray] *cis* is shape `(len(cols), 2)` where each row contains [lower, upper] for the given entry in *cols*

Notes

Confidence intervals are two-tailed. TODO: tails : string, optional

tails can be "two", "upper", or "lower"

Examples

```
>>> from numpy.random import standard_normal as stan
>>> from nipy.algorithms.statistics.models.regression import OLSModel
>>> x = np.hstack((stan((30,1)),stan((30,1)),stan((30,1))))
>>> beta=np.array([3.25, 1.5, 7.0])
>>> y = np.dot(x,beta) + stan((30))
>>> model = OLSModel(x).fit(y)
>>> confidence_intervals = model.conf_int(cols=(1,2))
```

logL()
The maximized log-likelihood

norm_resid()
Residuals, normalized to have unit length.

Notes

Is this supposed to return "standardized residuals," residuals standardized to have mean zero and approximately unit variance?

d_i = e_i / sqrt(MS_E)

Where MS_E = SSE / (n - k)

See: Montgomery and Peck 3.2.1 p. 68
Davidson and MacKinnon 15.2 p 662

predicted()
Return linear predictor values from a design matrix.

resid()
Residuals from the fit.

t(*column=None*)
Return the (Wald) t-statistic for a given parameter estimate.

Use Tcontrast for more complicated (Wald) t-statistics.

vcov(*matrix=None*, *column=None*, *dispersion=None*, *other=None*)
Variance/covariance matrix of linear contrast

Parameters

matrix: (dim, self.theta.shape[0]) array, optional
numerical contrast specification, where `dim` refers to the 'dimension' of the contrast i.e. 1 for t contrasts, 1 or more for F contrasts.

column: int, optional
alternative way of specifying contrasts (column index)

dispersion: float or (n_voxels,) array, optional
value(s) for the dispersion parameters

other: (dim, self.theta.shape[0]) array, optional
alternative contrast specification (?)

Returns

cov: (dim, dim) or (n_voxels, dim, dim) array
the estimated covariance matrix/matrices

Returns the variance/covariance matrix of a linear contrast of the estimates of theta, multiplied by *dispersion* which will often be an estimate of *dispersion*, like, sigma^2.
The covariance of interest is either specified as a (set of) column(s) or a matrix.

66.2.6 `WLSModel`

class `nipy.algorithms.statistics.models.regression.`**`WLSModel`**(*design*, *weights=1*)

Bases: *`OLSModel`*

A regression model with diagonal but non-identity covariance structure.

The weights are presumed to be (proportional to the) inverse of the variance of the observations.

Examples

```
>>> from nipy.algorithms.statistics.api import Term, Formula
>>> data = np.rec.fromarrays(([1,3,4,5,2,3,4], range(1,8)),
...                          names=('Y', 'X'))
>>> f = Formula([Term("X"), 1])
>>> dmtx = f.design(data, return_float=True)
>>> model = WLSModel(dmtx, weights=range(1,8))
>>> results = model.fit(data['Y'])
>>> results.theta
array([ 0.0952381 ,  2.91666667])
>>> results.t()
array([ 0.35684428,  2.0652652 ])
>>> print(results.Tcontrast([0,1]))
<T contrast: effect=2.91666666667, sd=1.41224801095, t=2.06526519708,
df_den=5>
>>> print(results.Fcontrast(np.identity(2)))
<F contrast: F=26.9986072423, df_den=5, df_num=2>
```

`__init__`(*design*, *weights=1*)

Parameters

design
[array-like] This is your design matrix. Data are assumed to be column ordered with observations in rows.

`fit`(*Y*)

Fit model to data *Y*

Full fit of the model including estimate of covariance matrix, (whitened) residuals and scale.

Parameters

Y
[array-like] The dependent variable for the Least Squares problem.

Returns

fit
[RegressionResults]

`has_intercept`()

Check if column of 1s is in column space of design

`information`(*beta*, *nuisance=None*)

Returns the information matrix at (beta, Y, nuisance).

See logL for details.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

nuisance
[dict] A dict with key 'sigma', which is an estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n` where n=Y.shape[0], X=self.design.

Returns

info
[array] The information matrix, the negative of the inverse of the Hessian of the of the log-likelihood function evaluated at (theta, Y, nuisance).

`initialize`(*design*)

Initialize (possibly re-initialize) a Model instance.

For instance, the design matrix of a linear model may change and some things must be recomputed.

`logL`(*beta*, *Y*, *nuisance=None*)

Returns the value of the loglikelihood function at beta.

Given the whitened design matrix, the loglikelihood is evaluated at the parameter vector, beta, for the dependent variable, Y and the nuisance parameter, sigma.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

Y
[ndarray] The dependent variable

nuisance
[dict, optional] A dict with key 'sigma', which is an optional estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n`, where n=Y.shape[0], X=self.design.

Returns

loglf
[float] The value of the loglikelihood function.

Notes

The log-Likelihood Function is defined as

$$\ell(\beta,\sigma,Y) = -\frac{n}{2}\log(2\pi\sigma^2) - \|Y-X\beta\|^2/(2\sigma^2)$$

The parameter σ above is what is sometimes referred to as a nuisance parameter. That is, the likelihood is considered as a function of β, but to evaluate it, a value of σ is needed.

If σ is not provided, then its maximum likelihood estimate:

$$\hat{\sigma}(\beta) = \frac{\text{SSE}(\beta)}{n}$$

is plugged in. This likelihood is now a function of only β and is technically referred to as a profile-likelihood.

References

[1]

`predict`(*design=None*)
After a model has been fit, results are (assumed to be) stored in self.results, which itself should have a predict method.

`rank`()
Compute rank of design matrix

`score`(*beta*, *Y*, *nuisance=None*)
Gradient of the loglikelihood function at (beta, Y, nuisance).

The graient of the loglikelihood function at (beta, Y, nuisance) is the score function.

See `logL()` for details.

Parameters

beta
[ndarray] The parameter estimates. Must be of length df_model.

Y
[ndarray] The dependent variable.

nuisance
[dict, optional] A dict with key 'sigma', which is an optional estimate of sigma. If None, defaults to its maximum likelihood estimate (with beta fixed) as `sum((Y - X*beta)**2) / n`, where n=Y.shape[0], X=self.design.

Returns

The gradient of the loglikelihood function.

whiten(*X*)

Whitener for WLS model, multiplies by sqrt(self.weights)

66.3 Functions

nipy.algorithms.statistics.models.regression.**ar_bias_correct**(*results*, *order*, *invM=None*)

Apply bias correction in calculating AR(p) coefficients from *results*

There is a slight bias in the rho estimates on residuals due to the correlations induced in the residuals by fitting a linear model. See [Worsley2002].

This routine implements the bias correction described in appendix A.1 of [Worsley2002].

Parameters

results
[ndarray or results object] If ndarray, assume these are residuals, from a simple model. If a results object, with attribute `resid`, then use these for the residuals. See Notes for more detail

order
[int] Order `p` of AR(p) model

invM
[None or array] Known bias correcting matrix for covariance. If None, calculate from `results.model`

Returns

rho
[array] Bias-corrected AR(p) coefficients

Notes

If *results* has attributes `resid` and `scale`, then assume `scale` has come from a fit of a potentially customized model, and we use that for the sum of squared residuals. In this case we also need `results.df_resid`. Otherwise we assume this is a simple Gaussian model, like OLS, and take the simple sum of squares of the residuals.

References

[Worsley2002]

nipy.algorithms.statistics.models.regression.**ar_bias_corrector**(*design*, *calc_beta*, *order=1*)

Return bias correcting matrix for *design* and AR order *order*

There is a slight bias in the rho estimates on residuals due to the correlations induced in the residuals by fitting a linear model. See [Worsley2002].

This routine implements the bias correction described in appendix A.1 of [Worsley2002].

Parameters

design
[array] Design matrix

calc_beta
[array] Moore-Penrose pseudoinverse of the (maybe) whitened design matrix. This is the matrix that, when applied to the (maybe whitened) data, produces the betas.

order
[int, optional] Order p of AR(p) process

Returns

invM
[array] Matrix to bias correct estimated covariance matrix in calculating the AR coefficients

References

[Worsley2002]

`nipy.algorithms.statistics.models.regression.`**`isestimable`**`(`*C*, *D*`)`

True if (Q, P) contrast *C* is estimable for (N, P) design *D*

From an Q x P contrast matrix *C* and an N x P design matrix *D*, checks if the contrast *C* is estimable by looking at the rank of `vstack([C,D])` and verifying it is the same as the rank of *D*.

Parameters

C
[(Q, P) array-like] contrast matrix. If *C* has is 1 dimensional assume shape (1, P)

D: (N, P) array-like
design matrix

Returns

tf
[bool] True if the contrast *C* is estimable on design *D*

Examples

```
>>> D = np.array([[1, 1, 1, 0, 0, 0],
...               [0, 0, 0, 1, 1, 1],
...               [1, 1, 1, 1, 1, 1]]).T
>>> isestimable([1, 0, 0], D)
False
>>> isestimable([1, -1, 0], D)
True
```

`nipy.algorithms.statistics.models.regression.`**`yule_walker`**`(`*X*, *order=1*, *method='unbiased'*, *df=None*, *inv=False*`)`

Estimate AR(p) parameters from a sequence X using Yule-Walker equation.

unbiased or maximum-likelihood estimator (mle)

See, for example:

http://en.wikipedia.org/wiki/Autoregressive_moving_average_model

Parameters

X
[ndarray of shape(n)]

order
[int, optional] Order of AR process.

method
[str, optional] Method can be “unbiased” or “mle” and this determines denominator in estimate of autocorrelation function (ACF) at lag k. If “mle”, the denominator is n=X.shape[0], if “unbiased” the denominator is n-k.

df
[int, optional] Specifies the degrees of freedom. If df is supplied, then it is assumed the X has df degrees of freedom rather than n.

inv
[bool, optional] Whether to return the inverse of the R matrix (see code)

Returns

rho
[(*order*,) ndarray]

sigma
[int] standard deviation of the residuals after fit

R_inv
[ndarray] If *inv* is True, also return the inverse of the R matrix

Notes

See also http://en.wikipedia.org/wiki/AR_model#Calculation_of_the_AR_parameters

CHAPTER

SIXTYSEVEN

ALGORITHMS.STATISTICS.MODELS.UTILS

67.1 Module: `algorithms.statistics.models.utils`

Inheritance diagram for `nipy.algorithms.statistics.models.utils`:

General matrix and other utilities for statistics

67.2 Class

67.3 `StepFunction`

class `nipy.algorithms.statistics.models.utils.`**`StepFunction`**(*x*, *y*, *ival=0.0*, *sorted=False*)

Bases: `object`

A basic step function

Values at the ends are handled in the simplest way possible: everything to the left of `x[0]` is set to *ival*; everything to the right of `x[-1]` is set to `y[-1]`.

Examples

```
>>> x = np.arange(20)
>>> y = np.arange(20)
>>> f = StepFunction(x, y)
>>>
>>> f(3.2)
3.0
>>> res = f([[3.2, 4.5],[24, -3.1]])
>>> np.all(res == [[ 3, 4],
...                [19, 0]])
True
```

`__init__`(*x*, *y*, *ival=0.0*, *sorted=False*)

67.4 Functions

`nipy.algorithms.statistics.models.utils.`**`ECDF`**(*values*)

Return the ECDF of an array as a step function.

`nipy.algorithms.statistics.models.utils.`**`mad`**(*a*, *c=0.6745*, *axis=0*)

Median Absolute Deviation:

median(abs(a - median(a))) / c

`nipy.algorithms.statistics.models.utils.`**`monotone_fn_inverter`**(*fn*, *x*, *vectorized=True*, ***keywords*)

Given a monotone function x (no checking is done to verify monotonicity) and a set of x values, return an linearly interpolated approximation to its inverse from its values on x.

CHAPTER

SIXTYEIGHT

ALGORITHMS.STATISTICS.ONESAMPLE

68.1 Module: `algorithms.statistics.onesample`

Utilities for one sample t-tests

68.2 Functions

`nipy.algorithms.statistics.onesample.`**`estimate_mean`**`(`*Y*, *sd*`)`

Estimate the mean of a sample given information about the standard deviations of each entry.

Parameters

Y

[ndarray] Data for which mean is to be estimated. Should have shape[0] == number of subjects.

sd

[ndarray] Standard deviation (subject specific) of the data for which the mean is to be estimated. Should have shape[0] == number of subjects.

Returns

value

[dict] This dictionary has keys ['effect', 'scale', 't', 'resid', 'sd']

`nipy.algorithms.statistics.onesample.`**`estimate_varatio`**`(`*Y*, *sd*, *df=None*, *niter=10*`)`

Estimate variance fixed/random effects variance ratio

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[0] == number of subjects.

sd

[array] Standard deviation (subject specific) of the data for which the mean is to be estimated. Should have shape[0] == number of subjects.

df

[int or None, optional] If supplied, these are used as weights when deriving the fixed effects variance. Should have length == number of subjects.

niter
[int, optional] Number of EM iterations to perform (default 10)

Returns

value
[dict] This dictionary has keys [‘fixed’, ‘ratio’, ‘random’], where ‘fixed’ 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’.

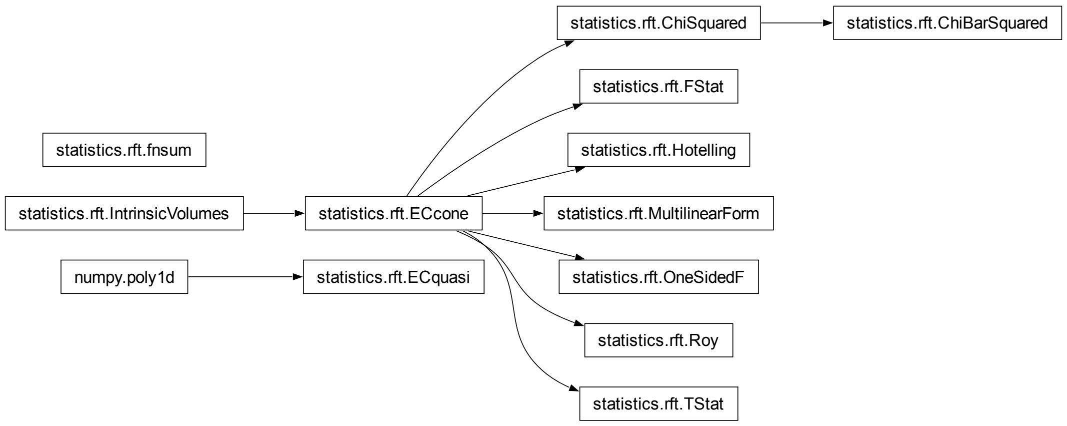
CHAPTER

SIXTYNINE

ALGORITHMS.STATISTICS.RFT

69.1 Module: `algorithms.statistics.rft`

Inheritance diagram for `nipy.algorithms.statistics.rft`:



Random field theory routines

The theoretical results for the EC densities appearing in this module were partially supported by NSF grant DMS-0405970.

Taylor, J.E. & Worsley, K.J. (2012). "Detecting sparse cone alternatives
for Gaussian random fields, with an application to fMRI". arXiv:1207.3840 [math.ST] and Statistica Sinica 23 (2013): 1629-1656.

Taylor, J.E. & Worsley, K.J. (2008). "Random fields of multivariate
test statistics, with applications to shape analysis." arXiv:0803.1708 [math.ST] and Annals of Statistics 36(2008): 1-27

69.2 Classes

69.2.1 `ChiBarSquared`

class `nipy.algorithms.statistics.rft.`**`ChiBarSquared`**(*dfn=1*, *search=[1]*)

Bases: `ChiSquared`

`__init__`(*dfn=1*, *search=[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 EC density in dimension *dim*

- ignoring a factor of (2pi)^{-(dim+1)/2} in front.

69.2.2 `ChiSquared`

class `nipy.algorithms.statistics.rft.`**`ChiSquared`**(*dfn*, *dfd=inf*, *search=[1]*)

Bases: `ECcone`

EC densities for a Chi-Squared(n) random field.

`__init__`(*dfn*, *dfd=inf*, *search=[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 EC density in dimension *dim*

- ignoring a factor of (2pi)^{-(dim+1)/2} in front.

69.2.3 `ECcone`

class `nipy.algorithms.statistics.rft.`**`ECcone`**(*mu=[1]*, *dfd=inf*, *search=[1]*, *product=[1]*)

Bases: `IntrinsicVolumes`

EC approximation to supremum distribution of var==1 Gaussian process

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 EC density in dimension *dim*

- ignoring a factor of (2pi)^{-(dim+1)/2} in front.

69.2.4 ECquasi

class nipy.algorithms.statistics.rft.**ECquasi**(*c_or_r*, *r=0*, *exponent=None*, *m=None*)

Bases: poly1d

Polynomials with premultiplier

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.

Examples

```
>>> import numpy
>>> from nipy.algorithms.statistics.rft import ECquasi
>>> x = numpy.linspace(0,1,101)
```

```
>>> a = ECquasi([3,4,5])
>>> a
ECquasi(array([3, 4, 5]), m=inf, exponent=0.000000)
>>> a(3) == 3*3**2 + 4*3 + 5
True
```

```
>>> b = ECquasi(a.coeffs, m=30, exponent=4)
>>> numpy.allclose(b(x), a(x) * numpy.power(1+x**2/30, -4))
True
```

__init__(*c_or_r*, *r=0*, *exponent=None*, *m=None*)

property c

The polynomial coefficients

change_exponent(*_pow*)

Change exponent

Multiply top and bottom by an integer multiple of the self.denom_poly.

Examples

```
>>> import numpy
>>> b = ECquasi([3,4,20], m=30, exponent=4)
>>> x = numpy.linspace(0,1,101)
>>> c = b.change_exponent(3)
>>> c
ECquasi(array([  1.11111111e-04,   1.48148148e-04,   1.07407407e-02,
         1.33333333e-02,   3.66666667e-01,   4.00000000e-01,
         5.00000000e+00,   4.00000000e+00,   2.00000000e+01]), m=30.000000,␣
↪exponent=7.000000)
>>> numpy.allclose(c(x), b(x))
True
```

property coef

The polynomial coefficients

property coefficients

The polynomial coefficients

property coeffs

The polynomial coefficients

compatible(*other*)

Check compatibility of degrees of freedom

Check whether the degrees of freedom of two instances are equal so that they can be multiplied together.

Examples

```
>>> import numpy
>>> b = ECquasi([3,4,20], m=30, exponent=4)
>>> x = numpy.linspace(0,1,101)
>>> c = b.change_exponent(3)
>>> b.compatible(c)
True
>>> d = ECquasi([3,4,20])
>>> b.compatible(d)
False
>>>
```

denom_poly()

Base of the premultiplier: (1+x^2/m).

Examples

```
>>> import numpy
>>> b = ECquasi([3,4,20], m=30, exponent=4)
>>> d = b.denom_poly()
>>> d
poly1d([ 0.03333333,  0.        ,  1.        ])
>>> numpy.allclose(d.c, [1./b.m,0,1])
True
```

deriv(*m=1*)

Evaluate derivative of ECquasi

Parameters

m
[int, optional]

Examples

```
>>> a = ECquasi([3,4,5])
>>> a.deriv(m=2)
ECquasi(array([6]), m=inf, exponent=0.000000)
```

```
>>> b = ECquasi([3,4,5], m=10, exponent=3)
>>> b.deriv()
ECquasi(array([-1.2, -2. ,  3. ,  4. ]), m=10.000000, exponent=4.000000)
```

integ(*m=1*, *k=0*)

Return an antiderivative (indefinite integral) of this polynomial.

Refer to *polyint* for full documentation.

See also:

polyint
equivalent function

property o

The order or degree of the polynomial

property order

The order or degree of the polynomial

property r

The roots of the polynomial, where self(x) == 0

property roots

The roots of the polynomial, where self(x) == 0

property variable

The name of the polynomial variable

69.2.5 FStat

class nipy.algorithms.statistics.rft.**FStat**(*dfn*, *dfd=inf*, *search=[1]*)

Bases: *ECcone*

EC densities for a F random field.

__init__(*dfn*, *dfd=inf*, *search=[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 EC density in dimension *dim*

- ignoring a factor of (2pi)^{-(dim+1)/2} in front.

69.2.6 Hotelling

class nipy.algorithms.statistics.rft.**Hotelling**(*dfd=inf*, *k=1*, *search=[1]*)

Bases: *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]*)

density(*x*, *dim*)

The EC density in dimension *dim*.

integ(*m=None*, *k=None*)

pvalue(*x*, *search=None*)

quasi(*dim*)

(Quasi-)polynomial parts of EC density in dimension *dim*

- ignoring a factor of (2pi)^{-(dim+1)/2} in front.

69.2.7 IntrinsicVolumes

class nipy.algorithms.statistics.rft.**IntrinsicVolumes**(*mu=[1]*)

Bases: object

Compute intrinsic volumes of products of sets

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]*)

69.2.8 MultilinearForm

class nipy.algorithms.statistics.rft.**MultilinearForm**(**dims*, ***keywords*)

Bases: *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*)

density(*x*, *dim*)

The EC density in dimension *dim*.

integ(*m=None*, *k=None*)

pvalue(*x*, *search=None*)

quasi(*dim*)

(Quasi-)polynomial parts of EC density in dimension *dim*

- ignoring a factor of (2pi)^{-(dim+1)/2} in front.

69.2.9 OneSidedF

class nipy.algorithms.statistics.rft.**OneSidedF**(*dfn*, *dfd=inf*, *search=[1]*)

Bases: *ECcone*

EC densities for one-sided F statistic

See:

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]*)

density(*x*, *dim*)

The EC density in dimension *dim*.

integ(*m=None*, *k=None*)

pvalue(*x*, *search=None*)

quasi(*dim*)

(Quasi-)polynomial parts of EC density in dimension *dim*

- ignoring a factor of (2pi)^{-(dim+1)/2} in front.

69.2.10 `Roy`

class `nipy.algorithms.statistics.rft.`**`Roy`**(*dfn=1*, *dfd=inf*, *k=1*, *search=[1]*)

Bases: *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]*)

`density`(*x*, *dim*)

The EC density in dimension *dim*.

`integ`(*m=None*, *k=None*)

`pvalue`(*x*, *search=None*)

`quasi`(*dim*)

(Quasi-)polynomial parts of EC density in dimension *dim*

- ignoring a factor of (2pi)^{-(dim+1)/2} in front.

69.2.11 `TStat`

class `nipy.algorithms.statistics.rft.`**`TStat`**(*dfd=inf*, *search=[1]*)

Bases: *ECcone*

EC densities for a t random field.

`__init__`(*dfd=inf*, *search=[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 EC density in dimension *dim*

- ignoring a factor of (2pi)^{-(dim+1)/2} in front.

69.2.12 `fnsum`

class `nipy.algorithms.statistics.rft.`**`fnsum`**(**items*)

Bases: `object`

`__init__`(**items*)

69.3 Functions

`nipy.algorithms.statistics.rft.Q`(*dim*, *dfd=inf*)

Q polynomial

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 [Worsley1994]

Parameters

dim
[int] dimension of polynomial

dfd
[scalar]

Returns

q_poly
[np.poly1d instance]

References

[Worsley1994]

`nipy.algorithms.statistics.rft.ball_search`(*n*, *r=1*)

A ball-shaped search region of radius r.

`nipy.algorithms.statistics.rft.binomial`(*n*, *k*)

Binomial coefficient

n!

c = ———

(n-k)! k!

Parameters

n
[float] n of (n, k)

k
[float] k of (n, k)

Returns

c
[float]

Examples

First 3 values of 4 th row of Pascal triangle

```
>>> [binomial(4, k) for k in range(3)]
[1.0, 4.0, 6.0]
```

nipy.algorithms.statistics.rft.**mu_ball**(*n*, *j*, *r=1*)

j`th curvature of `n-dimensional ball radius *r*

Return mu_j(B_n(r)), the j-th Lipschitz Killing curvature of the ball of radius r in R^n.

nipy.algorithms.statistics.rft.**mu_sphere**(*n*, *j*, *r=1*)

j`th curvature for `n dimensional sphere radius *r*

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.

nipy.algorithms.statistics.rft.**scale_space**(*region*, *interval*, *kappa=1.0*)

scale space intrinsic volumes of region x interval

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

nipy.algorithms.statistics.rft.**spherical_search**(*n*, *r=1*)

A spherical search region of radius r.

nipy.algorithms.statistics.rft.**volume2ball**(*vol*, *d=3*)

Approximate volume with ball

Approximate intrinsic volumes of a set with a given volume by those of a ball with a given dimension and equal volume.

CHAPTER

SEVENTY

ALGORITHMS.STATISTICS.UTILS

70.1 Module: `algorithms.statistics.utils`

70.2 Functions

`nipy.algorithms.statistics.utils.`**`check_cast_bin8`**`(`*arr*`)`

Return binary array *arr* as uint8 type, or raise if not binary.

Parameters

arr
[array-like]

Returns

bin8_arr
[uint8 array] *bin8_arr* has same shape as *arr*, is of dtype `np.uint8`, with values 0 and 1 only.

Raises

ValueError
When the array is not binary. Specifically, raise if, for any element `e`, `e != (e != 0)`.

`nipy.algorithms.statistics.utils.`**`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)]*`)`

Faces from simplices

Take a list of maximal simplices (by default a triangulation of a cube into 6 tetrahedra) and computes all faces

Parameters

maximal
[sequence of sequences, optional] Default is triangulation of cube into tetrahedra

Returns

faces
[dict]

`nipy.algorithms.statistics.utils.`**`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].

Parameters

center
[(d,) sequence of int, optional] Default is [0, 0, 0]

strides
[(d,) sequence of int, optional] Default is [4, 2, 1]. These are the strides given by `np.ones((2,2,2), np.bool_).strides`

Returns

complex
[dict] A dictionary with integer keys representing a simplicial complex. The vertices of the simplicial complex are the indices of the corners of the cube in a 'flattened' array with specified strides.

`nipy.algorithms.statistics.utils.`**`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.

`nipy.algorithms.statistics.utils.`**`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.

`nipy.algorithms.statistics.utils.`**`join_complexes`**(**complexes*)

Join a sequence of simplicial complexes.

Returns the union of all the particular faces.

`nipy.algorithms.statistics.utils.`**`multiple_fast_inv`**(*a*)

Compute the inverse of a set of arrays in-place

Parameters

a: array_like of shape (n_samples, M, M)
Set of square matrices to be inverted. *a* is changed in place.

Returns

a: ndarray shape (n_samples, M, M)
The input array *a*, overwritten with the inverses of the original 2D arrays in `a[0], a[1], ...`. Thus `a[0]` replaced with `inv(a[0])` etc.

Raises

LinAlgError
If *a* is singular.

ValueError
If *a* is not square, or not 2-dimensional.

Notes

This function is copied from scipy.linalg.inv, but with some customizations for speed-up from operating on multiple arrays. It also has some conditionals to work with different scipy versions.

`nipy.algorithms.statistics.utils.`**`multiple_mahalanobis`**`(`*effect*, *covariance*`)`

Returns the squared Mahalanobis distance for a given set of samples

Parameters

effect: array of shape (n_features, n_samples),
Each column represents a vector to be evaluated

covariance: array of shape (n_features, n_features, n_samples),
Corresponding covariance models stacked along the last axis

Returns

sqd: array of shape (n_samples,)
the squared distances (one per sample)

`nipy.algorithms.statistics.utils.`**`test_EC2`**`(`*shape*`)`

`nipy.algorithms.statistics.utils.`**`test_EC3`**`(`*shape*`)`

`nipy.algorithms.statistics.utils.`**`z_score`**`(`*pvalue*`)`

Return the z-score corresponding to a given p-value.

CHAPTER

SEVENTYONE

ALGORITHMS.UTILS.FAST_DISTANCE

71.1 Module: `algorithms.utils.fast_distance`

this module contains a function to perform fast distance computation on arrays

Author : Bertrand Thirion, 2008-2011

`nipy.algorithms.utils.fast_distance.`**`euclidean_distance`**(*X*, *Y=None*)

Considering the rows of X (and Y=X) as vectors, compute the distance matrix between each pair of vectors

Parameters

X, array of shape (n1,p)

Y=None, array of shape (n2,p)
if Y==None, then Y=X is used instead

Returns

ED, array of shape(n1, n2) with all the pairwise distance

CHAPTER

SEVENTYTWO

ALGORITHMS.UTILS.MATRICES

72.1 Module: `algorithms.utils.matrices`

Utilities for working with matrices

72.2 Functions

`nipy.algorithms.utils.matrices.`**`full_rank`**`(`*X*, *r=None*`)`

Return full-rank matrix whose column span is the same as X

Uses an SVD decomposition.

If the rank of *X* is known it can be specified by *r* – no check is made to ensure that this really is the rank of X.

Parameters

X
[array-like] 2D array which may not be of full rank.

r
[None or int] Known rank of *X*. r=None results in standard matrix rank calculation. We do not check *r* is really the rank of X; it is to speed up calculations when the rank is already known.

Returns

fX
[array] Full-rank matrix with column span matching that of *X*

`nipy.algorithms.utils.matrices.`**`matrix_rank`**`(`*M*, *tol=None*`)`

Return rank of matrix using SVD method

Rank of the array is the number of SVD singular values of the array that are greater than *tol*.

This version of matrix rank is very similar to the numpy.linalg version except for the use of:

- scipy.linalg.svd instead of numpy.linalg.svd.
- the MATLAB algorithm for default tolerance calculation

`matrix_rank` appeared in numpy.linalg in December 2009, first available in numpy 1.5.0.

Parameters

M
[array-like] array of <=2 dimensions

tol

[{None, float}] threshold below which SVD values are considered zero. If *tol* is None, and *S* is an array with singular values for *M*, and *eps* is the epsilon value for datatype of *S*, then *tol* set to `S.max() * eps * max(M.shape)`.

Notes

We check for numerical rank deficiency by using `tol=max(M.shape) * eps * S[0]` (where `S[0]` is the maximum singular value and thus the 2-norm of the matrix). This is one tolerance threshold for rank deficiency, and the default algorithm used by MATLAB **[#2]_**. When floating point roundoff is the main concern, then "numerical rank deficiency" is a reasonable choice. In some cases you may prefer other definitions. The most useful measure of the tolerance depends on the operations you intend to use on your matrix. For example, if your data come from uncertain measurements with uncertainties greater than floating point epsilon, choosing a tolerance near that uncertainty may be preferable. The tolerance may be absolute if the uncertainties are absolute rather than relative.

References

Baltimore: Johns Hopkins University Press, 1996. .. [#2] http://www.mathworks.com/help/techdoc/ref/rank.html

Examples

```
>>> matrix_rank(np.eye(4)) # Full rank matrix
4
>>> I=np.eye(4); I[-1,-1] = 0. # rank deficient matrix
>>> matrix_rank(I)
3
>>> matrix_rank(np.zeros((4,4))) # All zeros - zero rank
0
>>> matrix_rank(np.ones((4,))) # 1 dimension - rank 1 unless all 0
1
>>> matrix_rank(np.zeros((4,)))
0
>>> matrix_rank([1]) # accepts array-like
1
```

`nipy.algorithms.utils.matrices.`**`pos_recipr`**`(`*X*`)`

Return element-wise reciprocal of array, setting `X`<=0 to 0

Return the reciprocal of an array, setting all entries less than or equal to 0 to 0. Therefore, it presumes that X should be positive in general.

Parameters

X

[array-like]

Returns

rX

[array] array of same shape as *X*, dtype np.float64, with values set to 1/X where X > 0, 0 otherwise

`nipy.algorithms.utils.matrices.`**`recipr0`**`(`*`X`*`)`

Return element-wise reciprocal of array, `X`==0 -> 0

Return the reciprocal of an array, setting all entries equal to 0 as 0. It does not assume that X should be positive in general.

Parameters

X
[array-like]

Returns

rX
[array]

CHAPTER

SEVENTYTHREE

ALGORITHMS.UTILS.PCA

73.1 Module: `algorithms.utils.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.

73.2 Functions

`nipy.algorithms.utils.pca.`**`pca`**(*data*, *axis=0*, *mask=None*, *ncomp=None*, *standardize=True*, *design_keep=None*, *design_resid='mean'*, *tol_ratio=0.01*)

Compute the SVD PCA of an array-like thing over *axis*.

Parameters

data
[ndarray-like (float)] The array on which to perform PCA over axis *axis* (below)

axis
[int, optional] The axis over which to perform PCA (axis identifying observations). Default is 0 (first)

mask
[ndarray-like (**np.bool_**), optional] An optional mask, should have shape given by data axes, with *axis* removed, i.e.: `s = data.shape; s.pop(axis); msk_shape=s`

ncomp
[{None, int}, optional] How many component basis projections to return. If ncomp is None (the default) then the number of components is given by the calculated rank of the data, after applying *design_keep*, *design_resid* and *tol_ratio* below. We always return all the basis vectors and percent variance for each component; *ncomp* refers only to the number of basis_projections returned.

standardize
[bool, optional] If True, standardize so each time series (after application of *design_keep* and *design_resid*) has the same standard deviation, as calculated by the `np.std` function.

design_keep
[None or ndarray, optional] Data is projected onto the column span of design_keep. None (default) equivalent to `np.identity(data.shape[axis])`

design_resid
[str or None or ndarray, optional] After projecting onto the column span of design_keep, data is projected perpendicular to the column span of this matrix. If None, we do no such second projection. If a string 'mean', then the mean of the data is removed, equivalent to passing a column vector matrix of 1s.

tol_ratio
[float, optional] If `XZ` is the vector of singular values of the projection matrix from *design_keep* and *design_resid*, and S are the singular values of `XZ`, then *tol_ratio* is the value used to calculate the effective rank of the projection of the design, as in `rank = ((S / S.max) > tol_ratio).sum()`

Returns

results
[dict]

G is the number of non-trivial components found after applying

tol_ratio to the projections of *design_keep* and *design_resid*.

results has keys:

- **basis_vectors: series over *axis*, shape (data.shape[axis], G) -** the eigenvectors of the PCA
- **pcnt_var: percent variance explained by component, shape** (G,)
- **basis_projections: PCA components, with components varying** over axis *axis*; thus shape given by: `s = list(data.shape); s[axis] = ncomp`
- `axis`: axis over which PCA has been performed.

Notes

See `pca_image.m` from `fmristat` for Keith Worsley's code on which some of this is based.

See: http://en.wikipedia.org/wiki/Principal_component_analysis for some inspiration for naming - particularly 'basis_vectors' and 'basis_projections'

Examples

```
>>> arr = np.random.normal(size=(17, 10, 12, 14))
>>> msk = np.all(arr > -2, axis=0)
>>> res = pca(arr, mask=msk, ncomp=9)
```

Basis vectors are columns. There is one column for each component. The number of components is the calculated rank of the data matrix after applying the various projections listed in the parameters. In this case we are only removing the mean, so the number of components is one less than the axis over which we do the PCA (here axis=0 by default).

```
>>> res['basis_vectors'].shape
(17, 16)
```

Basis projections are arrays with components in the dimension over which we have done the PCA (axis=0 by default). Because we set *ncomp* above, we only retain *ncomp* components.

```
>>> res['basis_projections'].shape
(9, 10, 12, 14)
```

`nipy.algorithms.utils.pca.`**`pca_image`**(*img*, *axis='t'*, *mask=None*, *ncomp=None*, *standardize=True*, *design_keep=None*, *design_resid='mean'*, *tol_ratio=0.01*)

Compute the PCA of an image over a specified axis

Parameters

img
[Image] The image on which to perform PCA over the given *axis*

axis
[str or int, optional] Axis over which to perform PCA. Default is 't'. If *axis* is an integer, gives the index of the input (domain) axis of *img*. If *axis* is a str, can be an input (domain) name, or an output (range) name, that maps to an input (domain) name.

mask
[Image, optional] An optional mask, should have shape == image.shape[:3] and the same coordinate map as *img* but with *axis* dropped

ncomp
[{None, int}, optional] How many component basis projections to return. If ncomp is None (the default) then the number of components is given by the calculated rank of the data, after applying *design_keep*, *design_resid* and *tol_ratio* below. We always return all the basis vectors and percent variance for each component; *ncomp* refers only to the number of basis_projections returned.

standardize
[bool, optional] If True, standardize so each time series (after application of *design_keep* and *design_resid*) has the same standard deviation, as calculated by the `np.std` function.

design_keep
[None or ndarray, optional] Data is projected onto the column span of design_keep. None (default) equivalent to `np.identity(data.shape[axis])`

design_resid
[str or None or ndarray, optional] After projecting onto the column span of design_keep, data is projected perpendicular to the column span of this matrix. If None, we do no such second projection. If a string 'mean', then the mean of the data is removed, equivalent to passing a column vector matrix of 1s.

tol_ratio
[float, optional] If `XZ` is the vector of singular values of the projection matrix from *design_keep* and *design_resid*, and S are the singular values of `XZ`, then *tol_ratio* is the value used to calculate the effective rank of the projection of the design, as in `rank = ((S / S.max) > tol_ratio).sum()`

Returns

results
[dict]

L is the number of non-trivial components found after applying

tol_ratio to the projections of *design_keep* and *design_resid*.

results has keys: * `basis_vectors`: series over *axis*, shape (data.shape[axis], L) -

the eigenvectors of the PCA

- **pcnt_var: percent variance explained by component, shape**
  (L,)
- **basis_projections: PCA components, with components varying**
  over axis *axis*; thus shape given by: `s = list(data.shape); s[axis] = ncomp`
- `axis`: axis over which PCA has been performed.

Examples

```
>>> from nipy.testing import funcfile
>>> from nipy import load_image
>>> func_img = load_image(funcfile)
```

Time is the fourth axis

```
>>> func_img.coordmap.function_range
CoordinateSystem(coord_names=('aligned-x=L->R', 'aligned-y=P->A', 'aligned-z=I->S',
↪'t'), name='aligned', coord_dtype=float64)
>>> func_img.shape
(17, 21, 3, 20)
```

Calculate the PCA over time, by default

```
>>> res = pca_image(func_img)
>>> res['basis_projections'].coordmap.function_range
CoordinateSystem(coord_names=('aligned-x=L->R', 'aligned-y=P->A', 'aligned-z=I->S',
↪'PCA components'), name='aligned', coord_dtype=float64)
```

The number of components is one less than the number of time points

```
>>> res['basis_projections'].shape
(17, 21, 3, 19)
```

CHAPTER

SEVENTYFOUR

CLI.DIAGNOSE

74.1 Module: `cli.diagnose`

`nipy.cli.diagnose.`**`main`**`()`

CHAPTER

SEVENTYFIVE

CLI.IMG3DTO4D

75.1 Module: `cli.img3dto4d`

75.2 Functions

`nipy.cli.img3dto4d.`**`do_3d_to_4d`**(*filenames*, *check_affines=True*)

`nipy.cli.img3dto4d.`**`main`**()

CHAPTER

SEVENTYSIX

CLI.IMG4DTO3D

76.1 Module: `cli.img4dto3d`

`nipy.cli.img4dto3d.`**`main`**`()`

CHAPTER

SEVENTYSEVEN

CLI.REALIGN4D

77.1 Module: `cli.realign4d`

Command line wrapper of SpaceTimeRealign

Based on:

Alexis Roche (2011) A Four-Dimensional Registration Algorithm With Application to Joint Correction of Motion and Slice Timing in fMRI. IEEE Trans. Med. Imaging 30(8): 1546-1554

`nipy.cli.realign4d.`**`main`**`()`

CHAPTER

SEVENTYEIGHT

CLI.TSDIFFANA

78.1 Module: `cli.tsdiffana`

`nipy.cli.tsdiffana.`**`main`**`()`

CHAPTER

SEVENTYNINE

CONFTEST

79.1 Module: `conftest`

79.2 Functions

`nipy.conftest.`**`add_np`**`(`*doctest_namespace*`)`

`nipy.conftest.`**`in_tmp_path`**`()`

`nipy.conftest.`**`mpl_imports`**`()`

Force matplotlib to use agg backend for tests

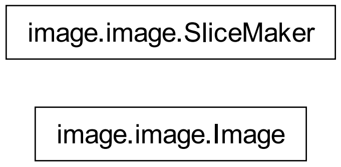
CHAPTER

EIGHTY

CORE.IMAGE.IMAGE

80.1 Module: `core.image.image`

Inheritance diagram for `nipy.core.image.image`:



Define the Image class and functions to work with Image instances

- fromarray : create an Image instance from an ndarray (deprecated in favor of using the Image constructor)
- subsample : slice an Image instance (deprecated in favor of image slicing)
- rollimg : roll an image axis to given location
- synchronized_order : match coordinate systems between images
- iter_axis : make iterator to iterate over an image axis
- is_image : test for an object obeying the Image API

80.2 Classes

80.2.1 `Image`

class `nipy.core.image.image.`**`Image`**(*data*, *coordmap*, *metadata=None*)

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 can be created through the module functions. See nipy.io for image IO such as `load` and `save`

Examples

Load an image from disk

```
>>> from nipy.testing import anatfile
>>> from nipy.io.api import load_image
>>> img = load_image(anatfile)
```

Make an image from an array. We need to make a meaningful coordinate map for the image.

```
>>> arr = np.zeros((21,64,64), dtype=np.int16)
>>> cmap = AffineTransform('kji', 'zxy', np.eye(4))
>>> img = Image(arr, cmap)
```

`__init__`(*data*, *coordmap*, *metadata=None*)

Create an *Image* object from array and *CoordinateMap* object.

Images are often created through the `load_image` function in the nipy base namespace.

Parameters

data
[array-like] object that as attribute `shape` and returns an array from `np.asarray(data)`

coordmap
[*AffineTransform* object] coordmap mapping the domain (input) voxel axes of the image to the range (reference, output) axes - usually mm in real world space

metadata
[dict, optional] Freeform metadata for image. Most common contents is `header` from nifti etc loaded images.

See also:

`load_image`
load `Image` from a file

`save_image`
save `Image` to a file

`affine`()

`axes`()

`coordmap = AffineTransform(function_domain=CoordinateSystem(coord_names=('i', 'j', 'k'), name='', coord_dtype=float64), function_range=CoordinateSystem(coord_names=('x', 'y', 'z'), name='', coord_dtype=float64), affine=array([[3., 0., 0., 0.], [0., 5., 0., 0.], [0., 0., 7., 0.], [0., 0., 0., 1.]]))`

`classmethod from_image`(*img*, *data=None*, *coordmap=None*, *metadata=None*)

Classmethod makes new instance of this *klass* from instance *img*

Parameters

data
[array-like] object that as attribute `shape` and returns an array from `np.asarray(data)`

coordmap
[*AffineTransform* object] coordmap mapping the domain (input) voxel axes of the image to the range (reference, output) axes - usually mm in real world space

metadata
[dict, optional] Freeform metadata for image. Most common contents is `header` from nifti etc loaded images.

Returns

img
[*klass* instance] New image with data from *data*, coordmap from *coordmap* maybe metadata from *metadata*

Notes

Subclasses of `Image` with different semantics for `__init__` will need to override this classmethod.

Examples

```
>>> from nipy import load_image
>>> from nipy.core.api import Image
>>> from nipy.testing import anatfile
>>> aimg = load_image(anatfile)
>>> arr = np.arange(24).reshape((2,3,4))
>>> img = Image.from_image(aimg, data=arr)
```

`get_fdata()`
Return data as a numpy array.

`property header`
The file header structure for this image, if available. This interface will soon go away - you should use ``img.metadata['header'] instead.

`metadata = {}`

`ndim()`

`reference()`

`renamed_axes(`*`**names_dict`***`)`**
Return a new image with input (domain) axes renamed

Axes renamed according to the input dictionary.

Parameters

****names_dict**
[dict] with keys being old names, and values being new names

Returns

newimg
[Image] An Image with the same data, having its axes renamed.

Examples

```
>>> data = np.random.standard_normal((11,9,4))
>>> im = Image(data, AffineTransform.from_params('ijk', 'xyz', np.identity(4),
↪'domain', 'range'))
>>> im_renamed = im.renamed_axes(i='slice')
>>> print(im_renamed.axes)
CoordinateSystem(coord_names=('slice', 'j', 'k'), name='domain', coord_
↪dtype=float64)
```

renamed_reference(***names_dict*)

Return new image with renamed output (range) coordinates

Coordinates renamed according to the dictionary

Parameters

****names_dict**
[dict] with keys being old names, and values being new names

Returns

newimg
[Image] An Image with the same data, having its output coordinates renamed.

Examples

```
>>> data = np.random.standard_normal((11,9,4))
>>> im = Image(data, AffineTransform.from_params('ijk', 'xyz', np.identity(4),
↪'domain', 'range'))
>>> im_renamed_reference = im.renamed_reference(x='newx', y='newy')
>>> print(im_renamed_reference.reference)
CoordinateSystem(coord_names=('newx', 'newy', 'z'), name='range', coord_
↪dtype=float64)
```

reordered_axes(*order=None*)

Return a new Image with reordered input coordinates.

This transposes the data as well.

Parameters

order
[None, sequence, optional] Sequence of int (giving indices) or str (giving names) - expressing new order of coordmap output coordinates. None (the default) results in reversed ordering.

Returns

r_img
[object] Image of same class as *self*, with reordered output coordinates.

Examples

```
>>> cmap = AffineTransform.from_start_step(
...             'ijk', 'xyz', [1, 2, 3], [4, 5, 6], 'domain', 'range')
>>> cmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k'), name='domain',␣
↪coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z'), name='range',␣
↪coord_dtype=float64),
   affine=array([[ 4.,  0.,  0.,  1.],
                 [ 0.,  5.,  0.,  2.],
                 [ 0.,  0.,  6.,  3.],
                 [ 0.,  0.,  0.,  1.]])
)
>>> im = Image(np.empty((30,40,50)), cmap)
>>> im_reordered = im.reordered_axes([2,0,1])
>>> im_reordered.shape
(50, 30, 40)
>>> im_reordered.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('k', 'i', 'j'), name='domain',␣
↪coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z'), name='range',␣
↪coord_dtype=float64),
   affine=array([[ 0.,  4.,  0.,  1.],
                 [ 0.,  0.,  5.,  2.],
                 [ 6.,  0.,  0.,  3.],
                 [ 0.,  0.,  0.,  1.]])
)
```

reordered_reference(*order=None*)

Return new Image with reordered output coordinates

New Image coordmap has reordered output coordinates. This does not transpose the data.

Parameters

order
[None, sequence, optional] sequence of int (giving indices) or str (giving names) - expressing new order of coordmap output coordinates. None (the default) results in reversed ordering.

Returns

r_img
[object] Image of same class as *self*, with reordered output coordinates.

Examples

```
>>> cmap = AffineTransform.from_start_step(
...             'ijk', 'xyz', [1, 2, 3], [4, 5, 6], 'domain', 'range')
>>> im = Image(np.empty((30,40,50)), cmap)
>>> im_reordered = im.reordered_reference([2,0,1])
>>> im_reordered.shape
(30, 40, 50)
>>> im_reordered.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k'), name='domain',␣
↪coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('z', 'x', 'y'), name='range',␣
↪coord_dtype=float64),
   affine=array([[ 0.,  0.,  6.,  3.],
                 [ 4.,  0.,  0.,  1.],
                 [ 0.,  5.,  0.,  2.],
                 [ 0.,  0.,  0.,  1.]])
)
```

shape()

80.2.2 SliceMaker

class nipy.core.image.image.**SliceMaker**

Bases: object

This class just creates slice objects for image resampling

It only has a __getitem__ method that returns its argument.

XXX Wouldn't need this if there was a way XXX to do this XXX subsample(img, [::2,::3,10:1:-1]) XXX XXX Could be something like this Subsample(img)[::2,::3,10:1:-1]

__init__(**args*, ***kwargs*)

80.3 Functions

nipy.core.image.image.**fromarray**(*data*, *innames*, *outnames*)

Create an image from array *data*, and input/output coordinate names

The mapping between the input and output coordinate names is the identity matrix.

Please don't use this routine, but instead prefer:

```
from nipy.core.api import Image, AffineTransform
img = Image(data, AffineTransform(innames, outnames, np.eye(4)))
```

where 4 is len(innames) + 1.

Parameters

data
[numpy array] A numpy array of three dimensions.

innames
[sequence] a list of input axis names

innames
[sequence] a list of output axis names

Returns

image
[An *Image* object]

See also:

`load`
function for loading images

`save`
function for saving images

Examples

```
>>> img = fromarray(np.zeros((2,3,4)), 'ijk', 'xyz')
>>> img.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k'), name='', coord_
↪dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z'), name='', coord_
↪dtype=float64),
   affine=array([[ 1.,  0.,  0.,  0.],
                 [ 0.,  1.,  0.,  0.],
                 [ 0.,  0.,  1.,  0.],
                 [ 0.,  0.,  0.,  1.]])
)
```

`nipy.core.image.image.`**`is_image`**`(`*obj*`)`

Returns true if this object obeys the Image API

This allows us to test for something that is duck-typing an image.

For now an array must have a 'coordmap' attribute, and a callable 'get_fdata' attribute.

Parameters

obj
[object] object for which to test API

Returns

is_img
[bool] True if object obeys image API

Examples

```
>>> from nipy.testing import anatfile
>>> from nipy.io.api import load_image
>>> img = load_image(anatfile)
>>> is_image(img)
True
>>> class C(object): pass
>>> c = C()
>>> is_image(c)
False
```

`nipy.core.image.image.`**`iter_axis`**(*img*, *axis*, *asarray=False*)

Return generator to slice an image *img* over *axis*

Parameters

img
[Image instance]

axis
[int or str] axis identifier, either name or axis number

asarray
[{False, True}, optional]

Returns

g
[generator] such that list(g) returns a list of slices over *axis*. If *asarray* is *False* the slices are images. If *asarray* is True, slices are the data from the images.

Examples

```
>>> data = np.arange(24).reshape((4,3,2))
>>> img = Image(data, AffineTransform('ijk', 'xyz', np.eye(4)))
>>> slices = list(iter_axis(img, 'j'))
>>> len(slices)
3
>>> slices[0].shape
(4, 2)
>>> slices = list(iter_axis(img, 'k', asarray=True))
>>> slices[1].sum() == data[:,:,1].sum()
True
```

`nipy.core.image.image.`**`rollaxis`**(*img*, *axis*, *inverse=False*)

rollaxis is deprecated! Please use rollimg instead

Roll *axis* backwards, until it lies in the first position.

It also reorders the reference coordinates by the same ordering. This is done to preserve a diagonal affine matrix if image.affine is diagonal. It also makes it possible to unambiguously specify an axis to roll along in terms of either a reference name (i.e. 'z') or an axis name (i.e. 'slice').

This function is deprecated; please use `rollimg` instead.

Parameters

img
: [Image] Image whose axes and reference coordinates are to be reordered by rolling.

axis
: [str or int] Axis to be rolled, can be specified by name or as an integer.

inverse
: [bool, optional] If inverse is True, then axis must be an integer and the first axis is returned to the position axis. This keyword is deprecated and we'll remove it in a future version of nipy.

Returns

newimg
: [Image] Image with reordered axes and reference coordinates.

Examples

```
>>> data = np.zeros((30,40,50,5))
>>> affine_transform = AffineTransform.from_params('ijkl', 'xyzt', np.diag([1,2,3,4,
↪1]))
>>> im = Image(data, affine_transform)
>>> im.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name='',␣
↪coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z', 't'), name='', coord_
↪dtype=float64),
   affine=array([[ 1.,  0.,  0.,  0.,  0.],
                 [ 0.,  2.,  0.,  0.,  0.],
                 [ 0.,  0.,  3.,  0.,  0.],
                 [ 0.,  0.,  0.,  4.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
>>> im_t_first = rollaxis(im, 't')
>>> np.diag(im_t_first.affine)
array([ 4.,  1.,  2.,  3.,  1.])
>>> im_t_first.shape
(5, 30, 40, 50)
>>> im_t_first.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('l', 'i', 'j', 'k'), name='',␣
↪coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('t', 'x', 'y', 'z'), name='', coord_
↪dtype=float64),
   affine=array([[ 4.,  0.,  0.,  0.,  0.],
                 [ 0.,  1.,  0.,  0.,  0.],
                 [ 0.,  0.,  2.,  0.,  0.],
                 [ 0.,  0.,  0.,  3.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
```

`nipy.core.image.image.`**`rollimg`**(*img*, *axis*, *start=0*, *fix0=True*)

Roll *axis* backwards in the inputs, until it lies before *start*

Parameters

img
[Image] Image whose axes and reference coordinates are to be reordered by rollimg.

axis
[str or int] Axis to be rolled, can be specified by name or as an integer. If an integer, axis is an input axis. If a name, can be name of input or output axis. If an output axis, we search for the closest matching input axis, and raise an AxisError if this fails.

start
[str or int, optional] position before which to roll axis *axis*. Default to 0. Can again be an integer (input axis) or name of input or output axis.

fix0
[bool, optional] Whether to allow for zero scaling when searching for an input axis matching an output axis. Useful for images where time scaling is 0.

Returns

newimg
[Image] Image with reordered input axes and corresponding data.

Examples

```
>>> data = np.zeros((30,40,50,5))
>>> affine_transform = AffineTransform('ijkl', 'xyzt', np.diag([1,2,3,4,1]))
>>> im = Image(data, affine_transform)
>>> im.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name='',␣
→coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z', 't'), name='', coord_
→dtype=float64),
   affine=array([[ 1.,  0.,  0.,  0.,  0.],
                 [ 0.,  2.,  0.,  0.,  0.],
                 [ 0.,  0.,  3.,  0.,  0.],
                 [ 0.,  0.,  0.,  4.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
>>> im_t_first = rollimg(im, 't')
>>> im_t_first.shape
(5, 30, 40, 50)
>>> im_t_first.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('l', 'i', 'j', 'k'), name='',␣
→coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z', 't'), name='', coord_
→dtype=float64),
   affine=array([[ 0.,  1.,  0.,  0.,  0.],
                 [ 0.,  0.,  2.,  0.,  0.],
                 [ 0.,  0.,  0.,  3.,  0.],
                 [ 4.,  0.,  0.,  0.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
```

`nipy.core.image.image.`**`subsample`**(*img*, *slice_object*)

Subsample an image

Please don't use this function, but use direct image slicing instead. That is, replace:

```
frame3 = subsample(im, slice_maker[:,:,:,3])
```

with:

```
frame3 = im[:,:,:,3]
```

Parameters

img
[Image]

slice_object: int, slice or sequence of slice
An object representing a numpy 'slice'.

Returns

img_subsampled: Image
An Image with data img.get_fdata()[slice_object] and an appropriately corrected CoordinateMap.

Examples

```
>>> from nipy.io.api import load_image
>>> from nipy.testing import funcfile
>>> from nipy.core.api import subsample, slice_maker
>>> im = load_image(funcfile)
>>> frame3 = subsample(im, slice_maker[:,:,:,3])
>>> np.allclose(frame3.get_fdata(), im.get_fdata()[:,:,:,3])
True
```

`nipy.core.image.image.`**`synchronized_order`**(*img*, *target_img*, *axes=True*, *reference=True*)

Reorder reference and axes of *img* to match target_img.

Parameters

img
[Image]

target_img
[Image]

axes
[bool, optional] If True, synchronize the order of the axes.

reference
[bool, optional] If True, synchronize the order of the reference coordinates.

Returns

newimg
[Image] An Image satisfying newimg.axes == target.axes (if axes == True), newimg.reference == target.reference (if reference == True).

Examples

```
>>> data = np.random.standard_normal((3,4,7,5))
>>> im = Image(data, AffineTransform.from_params('ijkl', 'xyzt', np.diag([1,2,3,4,
↪1])))
>>> im_scrambled = im.reordered_axes('iljk').reordered_reference('txyz')
>>> im == im_scrambled
False
>>> im_unscrambled = synchronized_order(im_scrambled, im)
>>> im == im_unscrambled
True
```

The images don't have to be the same shape

```
>>> data2 = np.random.standard_normal((3,11,9,4))
>>> im2 = Image(data, AffineTransform.from_params('ijkl', 'xyzt', np.diag([1,2,3,4,
↪1])))
>>> im_scrambled2 = im2.reordered_axes('iljk').reordered_reference('xtyz')
>>> im_unscrambled2 = synchronized_order(im_scrambled2, im)
>>> im_unscrambled2.coordmap == im.coordmap
True
```

or have the same coordmap

```
>>> data3 = np.random.standard_normal((3,11,9,4))
>>> im3 = Image(data3, AffineTransform.from_params('ijkl', 'xyzt', np.diag([1,9,3,-
↪2,1])))
>>> im_scrambled3 = im3.reordered_axes('iljk').reordered_reference('xtyz')
>>> im_unscrambled3 = synchronized_order(im_scrambled3, im)
>>> im_unscrambled3.axes == im.axes
True
>>> im_unscrambled3.reference == im.reference
True
>>> im_unscrambled4 = synchronized_order(im_scrambled3, im, axes=False)
>>> im_unscrambled4.axes == im.axes
False
>>> im_unscrambled4.axes == im_scrambled3.axes
True
>>> im_unscrambled4.reference == im.reference
True
```

CHAPTER

EIGHTYONE

CORE.IMAGE.IMAGE_LIST

81.1 Module: `core.image.image_list`

Inheritance diagram for `nipy.core.image.image_list`:

81.2 `ImageList`

class `nipy.core.image.image_list.`**`ImageList`**(*images=None*)

Bases: `object`

Class to contain ND image as list of (N-1)D images

`__init__`(*images=None*)

An implementation of a list of images.

Parameters

images

[iterable] an iterable object whose items are meant to be images; this is checked by asserting that each has a *coordmap* attribute and a `get_fdata` method. Note that Image objects are not iterable by default; use the `from_image` classmethod or `iter_axis` function to convert images to image lists - see examples below for the latter.

Examples

```
>>> from nipy.testing import funcfile
>>> from nipy.core.api import Image, ImageList, iter_axis
>>> from nipy.io.api import load_image
>>> funcim = load_image(funcfile)
>>> iterable_img = iter_axis(funcim, 't')
>>> ilist = ImageList(iterable_img)
>>> 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
>>> np.asarray(sublist).shape
(3, 17, 21, 3)
>>> newimg.get_fdata().shape
(17, 21, 3)
```

`classmethod from_image`(*image*, *axis=None*, *dropout=True*)

Create an image list from an *image* by slicing over *axis*

Parameters

image
[object] object with `coordmap` attribute

axis
[str or int] axis of *image* that should become the axis indexed by the image list.

dropout
[bool, optional] When taking slices from an image, we will leave an output dimension to the coordmap that has no corresponding input dimension. If *dropout* is True, drop this output dimension.

Returns

ilist
[`ImageList` instance]

`get_list_data`(*axis=None*)

Return data in ndarray with list dimension at position *axis*

Parameters

axis
[int] *axis* specifies which axis of the output will take the role of the list dimension. For example, 0 will put the list dimension in the first axis of the result.

Returns

data
[ndarray] data in image list as array, with data across elements of the list concetenated at dimension *axis* of the array.

Examples

```
>>> from nipy.testing import funcfile
>>> from nipy.io.api import load_image
>>> funcim = load_image(funcfile)
>>> ilist = ImageList.from_image(funcim, axis='t')
>>> ilist.get_list_data(axis=0).shape
(20, 17, 21, 3)
```

CHAPTER

EIGHTYTWO

CORE.IMAGE.IMAGE_SPACES

82.1 Module: `core.image.image_spaces`

Utilities for working with Images and common neuroimaging spaces

Images are very general things, and don't know anything about the kinds of spaces they refer to, via their coordinate map.

There are a set of common neuroimaging spaces. When we create neuroimaging Images, we want to place them in neuroimaging spaces, and return information about common neuroimaging spaces.

We do this by putting information about neuroimaging spaces in functions and variables in the `nipy.core.reference.spaces` module, and in this module.

This keeps the specific neuroimaging spaces out of our Image object.

```
>>> from nipy.core.api import Image, vox2mni, rollimg, xyz_affine, as_xyz_image
```

Make a standard 4D xyzt image in MNI space.

First the data and affine:

```
>>> data = np.arange(24, dtype=np.float32).reshape((1,2,3,4))
>>> affine = np.diag([2,3,4,1]).astype(float)
```

We can add the TR (==2.0) to make the full 5x5 affine we need

```
>>> img = Image(data, vox2mni(affine, 2.0))
>>> img.affine
array([[ 2.,  0.,  0.,  0.,  0.],
       [ 0.,  3.,  0.,  0.,  0.],
       [ 0.,  0.,  4.,  0.,  0.],
       [ 0.,  0.,  0.,  2.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

In this case the neuroimaging 'xyz_affine' is just the 4x4 from the 5x5 in the image

```
>>> xyz_affine(img)
array([[ 2.,  0.,  0.,  0.],
       [ 0.,  3.,  0.,  0.],
       [ 0.,  0.,  4.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

However, if we roll time first in the image array, we can't any longer get an xyz_affine that makes sense in relationship to the voxel data:

```
>>> img_t0 = rollimg(img, 't')
>>> xyz_affine(img_t0)
Traceback (most recent call last):
    ...
AxesError: First 3 input axes must correspond to X, Y, Z
```

But we can fix this:

```
>>> img_t0_affable = as_xyz_image(img_t0)
>>> xyz_affine(img_t0_affable)
array([[ 2.,  0.,  0.,  0.],
       [ 0.,  3.,  0.,  0.],
       [ 0.,  0.,  4.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

It also works with nibabel images, which can only have xyz_affines:

```
>>> import nibabel as nib
>>> nimg = nib.Nifti1Image(data, affine)
>>> xyz_affine(nimg)
array([[ 2.,  0.,  0.,  0.],
       [ 0.,  3.,  0.,  0.],
       [ 0.,  0.,  4.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

82.2 Functions

`nipy.core.image.image_spaces.`**`as_xyz_image`**(*img*, *name2xyz=None*)

Return version of *img* that has a valid xyz affine, or raise error

Parameters

img
[`Image` instance or nibabel image] It has a `coordmap` attribute (`Image`) or a `get_affine` method (nibabel image object)

name2xyz
[None or mapping] Object such that `name2xyz[ax_name]` returns 'x', or 'y' or 'z' or raises a KeyError for a str `ax_name`. None means use module default. Not used for nibabel *img* input.

Returns

reo_img
[`Image` instance or nibabel image] Returns image of same type as *img* input. If necessary, *reo_img* has its data and coordmap changed to allow it to return an xyz affine. If *img* is already xyz affable we return the input unchanged (`img is reo_img`).

Raises

SpaceTypeError
[if *img* does not have an affine coordinate map]

AxesError
[if not all of x, y, z recognized in *img* `coordmap` range]

AffineError
[if axes dropped from the affine contribute to x, y, z]

coordinates

`nipy.core.image.image_spaces.`**`is_xyz_affable`**(*img*, *name2xyz=None*)

Return True if the image *img* has an xyz affine

Parameters

img
[`Image` or nibabel `SpatialImage`] If `Image` test `img.coordmap`. If a nibabel image, return True

name2xyz
[None or mapping] Object such that `name2xyz[ax_name]` returns 'x', or 'y' or 'z' or raises a KeyError for a str `ax_name`. None means use module default. Not used for nibabel *img* input.

Returns

tf
[bool] True if *img* has an xyz affine, False otherwise

Examples

```
>>> from nipy.core.api import vox2mni, Image, rollimg
>>> arr = np.arange(24, dtype=np.float32).reshape((2,3,4,1))
>>> img = Image(arr, vox2mni(np.diag([2,3,4,5,1])))
>>> img.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name='voxels',
↪ coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-
↪z=I->S', 't'), name='mni', coord_dtype=float64),
   affine=array([[ 2.,  0.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.,  0.],
                 [ 0.,  0.,  4.,  0.,  0.],
                 [ 0.,  0.,  0.,  5.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
>>> is_xyz_affable(img)
True
>>> time0_img = rollimg(img, 't')
>>> time0_img.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('l', 'i', 'j', 'k'), name='voxels',
↪ coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-
↪z=I->S', 't'), name='mni', coord_dtype=float64),
   affine=array([[ 0.,  2.,  0.,  0.,  0.],
                 [ 0.,  0.,  3.,  0.,  0.],
                 [ 0.,  0.,  0.,  4.,  0.],
                 [ 5.,  0.,  0.,  0.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
```

(continues on next page)

(continued from previous page)

```
)
>>> is_xyz_affable(time0_img)
False
```

Nibabel images always have xyz affines

```
>>> import nibabel as nib
>>> nimg = nib.Nifti1Image(arr, np.diag([2,3,4,1]))
>>> is_xyz_affable(nimg)
True
```

`nipy.core.image.image_spaces.`**`make_xyz_image`**(*data*, *xyz_affine*, *world*, *metadata=None*)

Create 3D+ image embedded in space named in *world*

Parameters

data
[object] Object returning array from `np.asarray(obj)`, and having `shape` attribute. Should have at least 3 dimensions (`len(shape) >= 3`), and these three first 3 dimensions should be spatial

xyz_affine
[(4, 4) array-like or tuple] if (4, 4) array-like (the usual case), then an affine relating spatial dimensions in data (dimensions 0:3) to mm in XYZ space given in *world*. If a tuple, then contains two values: the (4, 4) array-like, and a sequence of scalings for the dimensions greater than 3. See examples.

world
[str or XYZSpace or CoordSysMaker or CoordinateSystem] World 3D space to which affine refers. See `spaces.get_world_cs()`

metadata
[None or mapping, optional] metadata for created image. Defaults to None, giving empty metadata.

Returns

img
[Image] image containing *data*, with coordmap constructed from *affine* and *world*, and with default voxel input coordinates. If the data has more than 3 dimensions, and you didn't specify the added zooms with a tuple *xyz_affine* parameter, the coordmap affine gets filled out with extra ones on the diagonal to give an (N+1, N+1) affine, with `N = len(data.shape)`

Examples

```
>>> data = np.arange(24).reshape((2, 3, 4))
>>> aff = np.diag([4, 5, 6, 1])
>>> img = make_xyz_image(data, aff, 'mni')
>>> img
Image(
  data=array([[[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]],
```

(continues on next page)

(continued from previous page)

```
              [[12, 13, 14, 15],
               [16, 17, 18, 19],
               [20, 21, 22, 23]]]]),
  coordmap=AffineTransform(
            function_domain=CoordinateSystem(coord_names=('i', 'j', 'k'), name=
↪'voxels', coord_dtype=float64),
            function_range=CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A',
↪ 'mni-z=I->S'), name='mni', coord_dtype=float64),
            affine=array([[ 4.,  0.,  0.,  0.],
                          [ 0.,  5.,  0.,  0.],
                          [ 0.,  0.,  6.,  0.],
                          [ 0.,  0.,  0.,  1.]])
         ))
```

Now make data 4D; we just add 1. to the diagonal for the new dimension

```
>>> data4 = data[..., None]
>>> img = make_xyz_image(data4, aff, 'mni')
>>> img.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name='voxels',
↪ coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-
↪z=I->S', 't'), name='mni', coord_dtype=float64),
   affine=array([[ 4.,  0.,  0.,  0.,  0.],
                 [ 0.,  5.,  0.,  0.,  0.],
                 [ 0.,  0.,  6.,  0.,  0.],
                 [ 0.,  0.,  0.,  1.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
```

We can pass in a scalar or tuple to specify scaling for the extra dimension

```
>>> img = make_xyz_image(data4, (aff, 2.0), 'mni')
>>> img.coordmap.affine
array([[ 4.,  0.,  0.,  0.,  0.],
       [ 0.,  5.,  0.,  0.,  0.],
       [ 0.,  0.,  6.,  0.,  0.],
       [ 0.,  0.,  0.,  2.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
>>> data5 = data4[..., None]
>>> img = make_xyz_image(data5, (aff, (2.0, 3.0)), 'mni')
>>> img.coordmap.affine
array([[ 4.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  5.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  6.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  2.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  3.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  1.]])
```

`nipy.core.image.image_spaces.`**`xyz_affine`**(*img*, *name2xyz=None*)

Return xyz affine from image *img* if possible, or raise error

Parameters

img
: [Image instance or nibabel image] It has a `coordmap` or attribute `affine` or method `get_affine`

name2xyz
: [None or mapping] Object such that `name2xyz[ax_name]` returns 'x', or 'y' or 'z' or raises a KeyError for a str `ax_name`. None means use module default. Not used for nibabel *img* input.

Returns

xyz_aff
: [(4,4) array] voxel to X, Y, Z affine mapping

Raises

SpaceTypeError
: [if *img* does not have an affine coordinate map]

AxesError
: [if not all of x, y, z recognized in *img* `coordmap` range]

AffineError
: [if axes dropped from the affine contribute to x, y, z]

coordinates

Examples

```
>>> from nipy.core.api import vox2mni, Image
>>> arr = np.arange(24).reshape((2,3,4,1)).astype(float)
>>> img = Image(arr, vox2mni(np.diag([2,3,4,5,1])))
>>> img.coordmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name='voxels',
↪ coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-
↪z=I->S', 't'), name='mni', coord_dtype=float64),
   affine=array([[ 2.,  0.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.,  0.],
                 [ 0.,  0.,  4.,  0.,  0.],
                 [ 0.,  0.,  0.,  5.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
>>> xyz_affine(img)
array([[ 2.,  0.,  0.,  0.],
       [ 0.,  3.,  0.,  0.],
       [ 0.,  0.,  4.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

Nibabel images always have xyz affines

```
>>> import nibabel as nib
>>> nimg = nib.Nifti1Image(arr, np.diag([2,3,4,1]))
>>> xyz_affine(nimg)
array([[ 2.,  0.,  0.,  0.],
```

(continues on next page)

(continued from previous page)

```
       [ 0.,  3.,  0.,  0.],
       [ 0.,  0.,  4.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

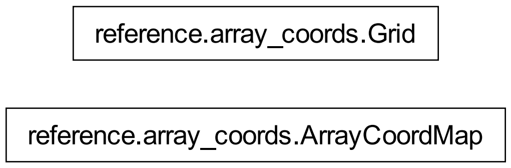
CHAPTER

EIGHTYTHREE

CORE.REFERENCE.ARRAY_COORDS

83.1 Module: `core.reference.array_coords`

Inheritance diagram for `nipy.core.reference.array_coords`:


reference.array_coords.Grid
reference.array_coords.ArrayCoordMap

Some CoordinateMaps have a domain 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.

83.2 Classes

83.2.1 `ArrayCoordMap`

class `nipy.core.reference.array_coords.`**`ArrayCoordMap`**(*coordmap*, *shape*)

Bases: `object`

Class combining coordinate map and array shape

When the function_domain 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 function_domain that are 'array' coordinates.

shape
[sequence of int] The size of the (implied) underlying array.

Examples

```
>>> aff = np.diag([0.6,1.1,2.3,1])
>>> aff[:3,3] = (0.1, 0.2, 0.3)
>>> cmap = AffineTransform.from_params('ijk', 'xyz', aff)
>>> cmap.ndims # number of (input, output) dimensions
(3, 3)
>>> acmap = ArrayCoordMap(cmap, (1, 2, 3))
```

Real world values at each array coordinate, one row per array coordinate (6 in this case), one column for each output dimension (3 in this case)

```
>>> acmap.values
array([[ 0.1,  0.2,  0.3],
       [ 0.1,  0.2,  2.6],
       [ 0.1,  0.2,  4.9],
       [ 0.1,  1.3,  0.3],
       [ 0.1,  1.3,  2.6],
       [ 0.1,  1.3,  4.9]])
```

Same values, but arranged in np.indices / np.mgrid format, first axis is for number of output coordinates (3 in our case), the rest are for the input shape (1, 2, 3)

```
>>> acmap.transposed_values.shape
(3, 1, 2, 3)
>>> acmap.transposed_values
array([[[[ 0.1,  0.1,  0.1],
         [ 0.1,  0.1,  0.1]]],


       [[[ 0.2,  0.2,  0.2],
         [ 1.3,  1.3,  1.3]]],


       [[[ 0.3,  2.6,  4.9],
         [ 0.3,  2.6,  4.9]]]])
```

static from_shape(*coordmap*, *shape*)
Create an evaluator assuming that coordmap.function_domain are 'array' coordinates.

property transposed_values
Get values of ArrayCoordMap in an array of shape (self.coordmap.ndims[1],) + self.shape)

property values
Get values of ArrayCoordMap in a 2-dimensional array of shape (product(self.shape), self.coordmap.ndims[1]))

83.2.2 Grid

class nipy.core.reference.array_coords.**Grid**(*coords*)

Bases: object

Simple class to construct AffineTransform 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.function_domain)
CoordinateSystem(coord_names=('i0', 'i1'), name='product', coord_dtype=float64)
>>> print(points.coordmap.function_range)
CoordinateSystem(coord_names=('x', 'y'), name='input', 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 AffineTransform instance is created with coords with identity transformation.

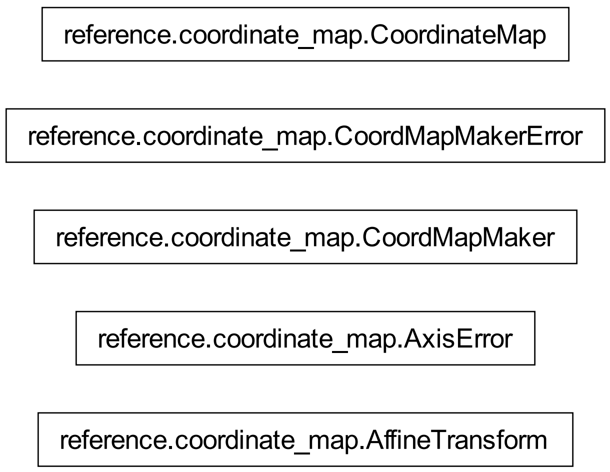
CHAPTER

EIGHTYFOUR

CORE.REFERENCE.COORDINATE_MAP

84.1 Module: `core.reference.coordinate_map`

Inheritance diagram for `nipy.core.reference.coordinate_map`:


reference.coordinate_map.CoordinateMap
reference.coordinate_map.CoordMapMakerError
reference.coordinate_map.CoordMapMaker
reference.coordinate_map.AxisError
reference.coordinate_map.AffineTransform

This module describes two types of *mappings*:

- CoordinateMap: a general function from a domain to a range, with a possible inverse function;
- AffineTransform: an affine function from a domain to a range, not necessarily of the same dimension, hence not always invertible.

Each of these objects is meant to encapsulate a tuple of (domain, range, function). Each of the mapping objects contain all the details about their domain CoordinateSystem, their range CoordinateSystem and the mapping between them.

84.1.1 Common API

They are separate classes, neither one inheriting from the other. They do, however, share some parts of an API, each having methods:

- renamed_domain : rename on the coordinates of the domain (returns a new mapping)
- renamed_range : rename the coordinates of the range (returns a new mapping)
- reordered_domain : reorder the coordinates of the domain (returns a new mapping)
- reordered_range : reorder the coordinates of the range (returns a new mapping)
- inverse : when appropriate, return the inverse *mapping*

These methods are implemented by module level functions of the same name.

They also share some attributes:

- ndims : the dimensions of the domain and range, respectively
- function_domain : CoordinateSystem describing the domain
- function_range : CoordinateSystem describing the range

84.1.2 Operations on mappings (module level functions)

- compose : Take a sequence of mappings (either CoordinateMaps or AffineTransforms) and return their composition. If they are all AffineTransforms, an AffineTransform is returned. This checks to ensure that domains and ranges of the various mappings agree.
- product : Take a sequence of mappings (either CoordinateMaps or AffineTransforms) and return a new mapping that has domain and range given by the concatenation of their domains and ranges, and the mapping simply concatenates the output of each of the individual mappings. If they are all AffineTransforms, an AffineTransform is returned. If they are all AffineTransforms that are in fact linear (i.e. origin=0) then can is represented as a block matrix with the size of the blocks determined by
- concat : Take a mapping and prepend a coordinate to its domain and range. For mapping `m`, this is the same as `product(AffineTransform.identity('concat'), m)`

84.2 Classes

84.2.1 `AffineTransform`

class `nipy.core.reference.coordinate_map.`**`AffineTransform`**(*function_domain*, *function_range*, *affine*)

Bases: `object`

Class for affine transformation from domain to a range

This class has an affine attribute, which is a matrix representing the affine transformation in homogeneous coordinates. This matrix is used to evaluate the function, rather than having an explicit function (as is the case for a CoordinateMap).

Examples

```
>>> inp_cs = CoordinateSystem('ijk')
>>> out_cs = CoordinateSystem('xyz')
>>> cm = AffineTransform(inp_cs, out_cs, np.diag([1, 2, 3, 1]))
>>> cm
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k'), name='', coord_
↪dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z'), name='', coord_
↪dtype=float64),
   affine=array([[ 1.,  0.,  0.,  0.],
                 [ 0.,  2.,  0.,  0.],
                 [ 0.,  0.,  3.,  0.],
                 [ 0.,  0.,  0.,  1.]])
)
```

```
>>> 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__`(*function_domain*, *function_range*, *affine*)

Initialize AffineTransform

Parameters

function_domain
[`CoordinateSystem`] input coordinates

function_range
[`CoordinateSystem`] output coordinates

affine
[array-like] affine homogeneous coordinate matrix

Notes

The dtype of the resulting matrix is determined by finding a safe typecast for the function_domain, function_range and affine.

`affine = array([[3, 0, 0, 0], [0, 4, 0, 0], [0, 0, 5, 0], [0, 0, 0, 1]])`

`static` **`from_params`**(*innames*, *outnames*, *params*, *domain_name=''*, *range_name=''*)

Create *AffineTransform* from *innames* and *outnames*

Parameters

innames
[sequence of str or str] The names of the axes of the domain. If str, then names given by `list(innames)`

outnames
[sequence of str or str] The names of the axes of the range. If str, then names given by `list(outnames)`

params
[AffineTransform, array or (array, array)] An affine function between the domain and range. This can be represented either by a single ndarray (which is interpreted as the representation of the function in homogeneous coordinates) or an (A,b) tuple.

domain_name
[str, optional] Name of domain CoordinateSystem

range_name
[str, optional] Name of range CoordinateSystem

Returns

aff
[`AffineTransform`]

Notes

Precondition
`len(shape) == len(names)`

Raises ValueError
`if len(shape) != len(names)`

`static from_start_step`(*innames*, *outnames*, *start*, *step*, *domain_name=''*, *range_name=''*)

New *AffineTransform* from names, start and step.

Parameters

innames
[sequence of str or str] The names of the axes of the domain. If str, then names given by `list(innames)`

outnames
[sequence of str or str] The names of the axes of the range. If str, then names given by `list(outnames)`

start
[sequence of float] Start vector used in constructing affine transformation

step
[sequence of float] Step vector used in constructing affine transformation

domain_name
[str, optional] Name of domain CoordinateSystem

range_name
[str, optional] Name of range CoordinateSystem

Returns

cm
[*CoordinateMap*]

Notes

len(names) == len(start) == len(step)

Examples

```
>>> cm = AffineTransform.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.]])
```

function_domain = CoordinateSystem(coord_names=('x',), name='', coord_dtype=float64)

function_range = CoordinateSystem(coord_names=('y',), name='', coord_dtype=float64)

static identity(*coord_names*, *name=''*)

Return an identity coordmap of the given shape

Parameters

coord_names
[sequence of str or str] The names of the axes of the domain. If str, then names given by `list(coord_names)`

name
[str, optional] Name of origin of coordinate system

Returns

cm
[*CoordinateMap*] `CoordinateMap` with `CoordinateSystem` domain and an identity transform, with identical domain and range.

Examples

```
>>> cm = AffineTransform.identity('ijk', 'somewhere')
>>> cm.affine
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> cm.function_domain
CoordinateSystem(coord_names=('i', 'j', 'k'), name='somewhere', coord_
↪dtype=float64)
>>> cm.function_range
CoordinateSystem(coord_names=('i', 'j', 'k'), name='somewhere', coord_
↪dtype=float64)
```

inverse(*preserve_dtype=False*)

Return coordinate map with inverse affine transform or None

Parameters

preserve_dtype
[bool] If False, return affine mapping from inverting the `affine`. The domain / range dtypes for the inverse may then change as a function of the dtype of the inverted `affine`. If True, try to invert our `affine`, and see if it can be cast to the needed data type, which is `self.function_domain.coord_dtype`. We need this dtype in order for the inverse to preserve the coordinate system dtypes.

Returns

aff_cm_inv
[`AffineTransform` instance or None] `AffineTransform` mapping from the *range* of input *self* to the *domain* of input *self* - the inverse of *self*. If `self.affine` was not invertible return None. If *preserve_dtype* is True, and the inverse of `self.affine` cannot be cast to `self.function_domain.coord_dtype`, then return None. Otherwise return `AffineTransform` inverse mapping. If *preserve_dtype* is False, the domain / range dtypes of the return inverse may well be different from those of the input *self*.

Examples

```
>>> input_cs = CoordinateSystem('ijk', coord_dtype=np.int_)
>>> output_cs = CoordinateSystem('xyz', coord_dtype=np.int_)
>>> affine = np.array([[1,0,0,1],
...                    [0,1,0,1],
...                    [0,0,1,1],
...                    [0,0,0,1]])
>>> affine_transform = AffineTransform(input_cs, output_cs, affine)
>>> affine_transform([2,3,4])
array([3, 4, 5])
```

The inverse transform, by default, generates a floating point inverse matrix and therefore floating point output:

```
>>> affine_transform_inv = affine_transform.inverse()
>>> affine_transform_inv([2, 6, 12])
array([  1.,   5.,  11.])
```

You can force it to preserve the coordinate system dtype with the *preserve_dtype* flag:

```
>>> at_inv_preserved = affine_transform.inverse(preserve_dtype=True)
>>> at_inv_preserved([2, 6, 12])
array([ 1,   5,  11])
```

If you *preserve_dtype*, and there is no inverse affine preserving the dtype, the inverse is None:

```
>>> affine2 = affine.copy()
>>> affine2[0, 0] = 2 # now inverse can't be integer
>>> aff_t = AffineTransform(input_cs, output_cs, affine2)
>>> aff_t.inverse(preserve_dtype=True) is None
True
```

ndims = (3, 3)

renamed_domain(*newnames*, *name=''*)
New AffineTransform with function_domain renamed

Parameters

newnames
[dict] A dictionary whose keys are integers or are in mapping.function_domain.coord_names and whose values are the new names.

Returns

newmapping
[AffineTransform] A new AffineTransform with renamed function_domain.

Examples

```
>>> affine_domain = CoordinateSystem('ijk')
>>> affine_range = CoordinateSystem('xyz')
>>> affine_matrix = np.identity(4)
>>> affine_mapping = AffineTransform(affine_domain, affine_range, affine_matrix)
```

```
>>> new_affine_mapping = affine_mapping.renamed_domain({'i':'phase','k':'freq',
↪'j':'slice'})
>>> new_affine_mapping.function_domain
CoordinateSystem(coord_names=('phase', 'slice', 'freq'), name='', coord_
↪dtype=float64)
```

```
>>> new_affine_mapping = affine_mapping.renamed_domain({'i':'phase','k':'freq',
↪'l':'slice'})
Traceback (most recent call last):
   ...
ValueError: no domain coordinate named l
```

renamed_range(*newnames*, *name=''*)

New AffineTransform with renamed function_domain

Parameters

newnames
[dict] A dictionary whose keys are integers or are in mapping.function_range.coord_names and whose values are the new names.

Returns

newmapping
[AffineTransform] A new AffineTransform with renamed function_range.

Examples

```
>>> affine_domain = CoordinateSystem('ijk')
>>> affine_range = CoordinateSystem('xyz')
>>> affine_matrix = np.identity(4)
>>> affine_mapping = AffineTransform(affine_domain, affine_range, affine_matrix)
```

```
>>> new_affine_mapping = affine_mapping.renamed_range({'x':'u'})
>>> new_affine_mapping.function_range
CoordinateSystem(coord_names=('u', 'y', 'z'), name='', coord_dtype=float64)
```

```
>>> new_affine_mapping = affine_mapping.renamed_range({'w':'u'})
Traceback (most recent call last):
   ...
ValueError: no range coordinate named w
```

reordered_domain(*order=None*)

New AffineTransform with function_domain reordered

Default behaviour is to reverse the order of the coordinates.

Parameters

order
[sequence] Order to use, defaults to reverse. The elements can be integers, strings or 2-tuples of strings. If they are strings, they should be in mapping.function_domain.coord_names.

Returns

newmapping :AffineTransform
A new AffineTransform with the coordinates of function_domain reordered.

Examples

```
>>> input_cs = CoordinateSystem('ijk')
>>> output_cs = CoordinateSystem('xyz')
>>> cm = AffineTransform(input_cs, output_cs, np.identity(4))
>>> cm.reordered_domain('ikj').function_domain
CoordinateSystem(coord_names=('i', 'k', 'j'), name='', coord_dtype=float64)
```

reordered_range(*order=None*)

New AffineTransform with function_range reordered

Defaults to reversing the coordinates of function_range.

Parameters

order
[sequence] Order to use, defaults to reverse. The elements can be integers, strings or 2-tuples of strings. If they are strings, they should be in mapping.function_range.coord_names.

Returns

newmapping
[AffineTransform] A new AffineTransform with the coordinates of function_range reordered.

Examples

```
>>> input_cs = CoordinateSystem('ijk')
>>> output_cs = CoordinateSystem('xyz')
>>> cm = AffineTransform(input_cs, output_cs, np.identity(4))
>>> cm.reordered_range('xzy').function_range
CoordinateSystem(coord_names=('x', 'z', 'y'), name='', coord_dtype=float64)
>>> cm.reordered_range([0,2,1]).function_range.coord_names
('x', 'z', 'y')
```

```
>>> newcm = cm.reordered_range('yzx')
>>> newcm.function_range.coord_names
('y', 'z', 'x')
```

similar_to(*other*)

Does *other* have similar coordinate systems and same mappings?

A "similar" coordinate system is one with the same coordinate names and data dtype, but ignoring the coordinate system name.

84.2.2 AxisError

class nipy.core.reference.coordinate_map.**AxisError**

Bases: `Exception`

Error for incorrect axis selection

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

84.2.3 CoordMapMaker

class nipy.core.reference.coordinate_map.**CoordMapMaker**(*domain_maker*, *range_maker*)

Bases: `object`

Class to create coordinate maps of different dimensions

__init__(*domain_maker*, *range_maker*)

Create coordinate map maker

Parameters

domain_maker
[callable] A coordinate system maker, returning a coordinate system with input argument only `N`, an integer giving the length of the coordinate map.

range_maker
[callable] A coordinate system maker, returning a coordinate system with input argument only `N`, an integer giving the length of the coordinate map.

Examples

```
>>> from nipy.core.reference.coordinate_system import CoordSysMaker
>>> dmaker = CoordSysMaker('ijkl', 'generic-array')
>>> rmaker = CoordSysMaker('xyzt', 'generic-scanner')
>>> cm_maker = CoordMapMaker(dmaker, rmaker)
```

affine_maker

alias of *AffineTransform*

generic_maker

alias of *CoordinateMap*

make_affine(*affine*, *append_zooms=()*, *append_offsets=()*)

Create affine coordinate map

Parameters

affine
[(M, N) array-like] Array expressing the affine transformation

append_zooms
[scalar or sequence length E] If scalar, converted to sequence length E==1. Append E entries to the diagonal of *affine* (see examples)

append_offsets
[scalar or sequence length F] If scalar, converted to sequence length F==1. If F==0, and E!=0, use sequence of zeros length E. Append E entries to the translations (final column) of *affine* (see examples).

Returns

affmap
[`AffineTransform` coordinate map]

Examples

```
>>> from nipy.core.reference.coordinate_system import CoordSysMaker
>>> dmaker = CoordSysMaker('ijkl', 'generic-array')
>>> rmaker = CoordSysMaker('xyzt', 'generic-scanner')
>>> cm_maker = CoordMapMaker(dmaker, rmaker)
>>> cm_maker.make_affine(np.diag([2,3,4,1]))
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k'), name='generic-
↪array', coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z'), name='generic-
↪scanner', coord_dtype=float64),
   affine=array([[ 2.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.],
                 [ 0.,  0.,  4.,  0.],
                 [ 0.,  0.,  0.,  1.]])
)
```

We can add extra orthogonal dimensions, by specifying the diagonal elements:

```
>>> cm_maker.make_affine(np.diag([2,3,4,1]), 6)
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name=
↪'generic-array', coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z', 't'), name=
↪'generic-scanner', coord_dtype=float64),
   affine=array([[ 2.,  0.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.,  0.],
                 [ 0.,  0.,  4.,  0.,  0.],
                 [ 0.,  0.,  0.,  6.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
```

Or the diagonal elements and the offset elements:

```
>>> cm_maker.make_affine(np.diag([2,3,4,1]), [6], [9])
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name=
↪'generic-array', coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z', 't'), name=
↪'generic-scanner', coord_dtype=float64),
   affine=array([[ 2.,  0.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.,  0.],
                 [ 0.,  0.,  4.,  0.,  0.],
                 [ 0.,  0.,  0.,  6.,  9.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
```

make_cmap(*domain_N*, *xform*, *inv_xform=None*)

Coordinate map with transform function *xform*

Parameters

domain_N
[int] Number of domain coordinates

xform
[callable] Function that transforms points of dimension *domain_N*

inv_xform
[None or callable, optional] Function, such that `inv_xform(xform(pts))` returns `pts`

Returns

cmap
[`CoordinateMap`]

Examples

```
>>> from nipy.core.reference.coordinate_system import CoordSysMaker
>>> dmaker = CoordSysMaker('ijkl', 'generic-array')
>>> rmaker = CoordSysMaker('xyzt', 'generic-scanner')
>>> cm_maker = CoordMapMaker(dmaker, rmaker)
>>> cm_maker.make_cmap(4, lambda x : x+1)
CoordinateMap(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name=
↪'generic-array', coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('x', 'y', 'z', 't'), name=
↪'generic-scanner', coord_dtype=float64),
   function=<function <lambda> at ...>
  )
```

84.2.4 CoordMapMakerError

class nipy.core.reference.coordinate_map.**CoordMapMakerError**

Bases: Exception

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

84.2.5 CoordinateMap

class nipy.core.reference.coordinate_map.**CoordinateMap**(*function_domain*, *function_range*, *function*, *inverse_function=None*)

Bases: object

A set of domain and range CoordinateSystems and a function between them.

For example, the function may represent the mapping of a voxel (the domain of the function) to real space (the range). The function may be an affine or non-affine transformation.

Examples

```
>>> function_domain = CoordinateSystem('ijk', 'voxels')
>>> function_range = CoordinateSystem('xyz', 'world')
>>> mni_orig = np.array([-90.0, -126.0, -72.0])
>>> function = lambda x: x + mni_orig
>>> inv_function = lambda x: x - mni_orig
>>> cm = CoordinateMap(function_domain, function_range, function, inv_function)
```

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.function(x)
array([[ -90., -126.,  -72.],
       [ -89., -126.,  -72.],
       [ -88., -126.,  -72.]])
```

```
>>> x = CoordinateSystem('x')
>>> y = CoordinateSystem('y')
>>> m = CoordinateMap(x, y, np.exp, np.log)
>>> m
CoordinateMap(
   function_domain=CoordinateSystem(coord_names=('x',), name='', coord_
↪dtype=float64),
   function_range=CoordinateSystem(coord_names=('y',), name='', coord_
↪dtype=float64),
   function=<ufunc 'exp'>,
   inverse_function=<ufunc 'log'>
  )
>>> m.inverse()
CoordinateMap(
   function_domain=CoordinateSystem(coord_names=('y',), name='', coord_
↪dtype=float64),
   function_range=CoordinateSystem(coord_names=('x',), name='', coord_
↪dtype=float64),
   function=<ufunc 'log'>,
   inverse_function=<ufunc 'exp'>
  )
```

Attributes

function_domain
[`CoordinateSystem` instance] The input coordinate system.

function_range
[`CoordinateSystem` instance] The output coordinate system.

function
[callable] exp(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

inverse_function
[None or callable] log(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

`__init__`(*function_domain*, *function_range*, *function*, *inverse_function=None*)

Create a CoordinateMap given function, domain and range.

Parameters

function_domain
[`CoordinateSystem`] The input coordinate system.

function_range
[`CoordinateSystem`] The output coordinate system

function
[callable] The function between function_domain and function_range. It should be a callable that accepts arrays of shape (N, function_domain.ndim) and returns arrays of shape (N, function_range.ndim), where N is the number of points for transformation.

inverse_function
[None or callable, optional] The optional inverse of function, with the intention being `x = inverse_function(function(x))`. If the function is affine and invertible, then this is true for all x. The default is None

Returns

coordmap
[CoordinateMap]

function(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) = <ufunc 'exp'>

function_domain = CoordinateSystem(coord_names=('x',), name='', coord_dtype=float64)

function_range = CoordinateSystem(coord_names=('y',), name='', coord_dtype=float64)

inverse()
New CoordinateMap with the functions reversed

inverse_function(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) = <ufunc 'log'>

ndims = (1, 1)

renamed_domain(*newnames*, *name=''*)
New CoordinateMap with function_domain renamed

Parameters

newnames
[dict] A dictionary whose keys are integers or are in mapping.function_domain.coord_names and whose values are the new names.

Returns

newmaping
[CoordinateMap] A new CoordinateMap with renamed function_domain.

Examples

```
>>> domain = CoordinateSystem('ijk')
>>> range = CoordinateSystem('xyz')
>>> cm = CoordinateMap(domain, range, lambda x:x+1)
```

```
>>> new_cm = cm.renamed_domain({'i':'phase','k':'freq','j':'slice'})
>>> new_cm.function_domain
CoordinateSystem(coord_names=('phase', 'slice', 'freq'), name='', coord_
↪dtype=float64)
```

```
>>> new_cm = cm.renamed_domain({'i':'phase','k':'freq','l':'slice'})
Traceback (most recent call last):
   ...
ValueError: no domain coordinate named l
```

renamed_range(*newnames*, *name=''*)

New CoordinateMap with function_domain renamed

Parameters

newnames
[dict] A dictionary whose keys are integers or are in mapping.function_range.coord_names and whose values are the new names.

Returns

newmapping
[CoordinateMap] A new CoordinateMap with renamed function_range.

Examples

```
>>> domain = CoordinateSystem('ijk')
>>> range = CoordinateSystem('xyz')
>>> cm = CoordinateMap(domain, range, lambda x:x+1)
```

```
>>> new_cm = cm.renamed_range({'x':'u'})
>>> new_cm.function_range
CoordinateSystem(coord_names=('u', 'y', 'z'), name='', coord_dtype=float64)
```

```
>>> new_cm = cm.renamed_range({'w':'u'})
Traceback (most recent call last):
   ...
ValueError: no range coordinate named w
```

reordered_domain(*order=None*)

Create a new CoordinateMap with the coordinates of function_domain reordered. Default behaviour is to reverse the order of the coordinates.

Parameters

order
[sequence] Order to use, defaults to reverse. The elements can be integers, strings or 2-tuples of strings. If they are strings, they should be in mapping.function_domain.coord_names.

Returns

newmapping
[CoordinateMap] A new CoordinateMap with the coordinates of function_domain reordered.

Examples

```
>>> input_cs = CoordinateSystem('ijk')
>>> output_cs = CoordinateSystem('xyz')
>>> cm = CoordinateMap(input_cs, output_cs, lambda x:x+1)
>>> cm.reordered_domain('ikj').function_domain
CoordinateSystem(coord_names=('i', 'k', 'j'), name='', coord_dtype=float64)
```

reordered_range(*order=None*)

Nnew CoordinateMap with function_range reordered.

Defaults to reversing the coordinates of function_range.

Parameters

order
[sequence] Order to use, defaults to reverse. The elements can be integers, strings or 2-tuples of strings. If they are strings, they should be in mapping.function_range.coord_names.

Returns

newmapping
[CoordinateMap] A new CoordinateMap with the coordinates of function_range reordered.

Examples

```
>>> input_cs = CoordinateSystem('ijk')
>>> output_cs = CoordinateSystem('xyz')
>>> cm = CoordinateMap(input_cs, output_cs, lambda x:x+1)
>>> cm.reordered_range('xzy').function_range
CoordinateSystem(coord_names=('x', 'z', 'y'), name='', coord_dtype=float64)
>>> cm.reordered_range([0,2,1]).function_range.coord_names
('x', 'z', 'y')
```

```
>>> newcm = cm.reordered_range('yzx')
>>> newcm.function_range.coord_names
('y', 'z', 'x')
```

similar_to(*other*)

Does *other* have similar coordinate systems and same mappings?

A “similar” coordinate system is one with the same coordinate names and data dtype, but ignoring the coordinate system name.

84.3 Functions

`nipy.core.reference.coordinate_map.`**`append_io_dim`**(*cm*, *in_name*, *out_name*, *start=0*, *step=1*)

Append input and output dimension to coordmap

Parameters

cm
[Affine] Affine coordinate map instance to which to append dimension

in_name
[str] Name for new input dimension

out_name
[str] Name for new output dimension

start
[float, optional] Offset for transformed values in new dimension

step
[float, optional] Step, or scale factor for transformed values in new dimension

Returns

cm_plus
[Affine] New coordinate map with appended dimension

Examples

Typical use is creating a 4D coordinate map from a 3D

```
>>> cm3d = AffineTransform.from_params('ijk', 'xyz', np.diag([1,2,3,1]))
>>> cm4d = append_io_dim(cm3d, 'l', 't', 9, 5)
>>> cm4d.affine
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  2.,  0.,  0.,  0.],
       [ 0.,  0.,  3.,  0.,  0.],
       [ 0.,  0.,  0.,  5.,  9.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

`nipy.core.reference.coordinate_map.`**`axmap`**(*coordmap*, *direction='in2out'*, *fix0=True*)

Return mapping between input and output axes

Parameters

coordmap
[Affine] Affine coordinate map instance for which to get axis mappings

direction
[{'in2out', 'out2in', 'both'}] direction to find mapping. If 'in2out', returned mapping will have keys from the input axis (names and indices) and values of corresponding output axes. If 'out2in' the keys will be output axis names, indices and the values will be input axis indices. If both, return both mappings.

fix0: bool, optional
Whether to fix potential 0 TR in affine

Returns

map
[dict or tuple]

- if *direction* == 'in2out' - mapping with keys of input names and input indices, values of output indices. Mapping is to closest matching axis. None means there appears to be no matching axis
- if *direction* == 'out2in' - mapping with keys of output names and input indices, values of input indices, as above.
- if *direction* == 'both' - tuple of (input to output mapping, output to input mapping)

`nipy.core.reference.coordinate_map.`**`compose`**`(`**cmaps*`)`

Return the composition of two or more CoordinateMaps.

Parameters

cmaps
[sequence of CoordinateMaps]

Returns

cmap
[`CoordinateMap`] The resulting CoordinateMap has function_domain == cmaps[-1].function_domain and function_range == cmaps[0].function_range

Examples

```
>>> cmap = AffineTransform.from_params('i', 'x', np.diag([2.,1.]))
>>> cmapi = cmap.inverse()
>>> id1 = compose(cmap,cmapi)
>>> id1.affine
array([[ 1.,  0.],
       [ 0.,  1.]])
```

```
>>> id2 = compose(cmapi,cmap)
>>> id1.function_domain.coord_names
('x',)
>>> id2.function_domain.coord_names
('i',)
```

`nipy.core.reference.coordinate_map.`**`drop_io_dim`**`(`*cm*, *axis_id*, *fix0=True*`)`

Drop dimensions *axis_id* from coordinate map, if orthogonal to others

If you specify an input dimension, drop that dimension and any corresponding output dimension, as long as all other outputs are orthogonal to dropped input. If you specify an output dimension, drop that dimension and any corresponding input dimension, as long as all other inputs are orthogonal to dropped output.

Parameters

cm
[class:*AffineTransform*] Affine coordinate map instance

axis_id
[int or str] If int, gives index of *input* axis to drop. If str, gives name of input *or* output axis to drop. When specifying an input axis: if given input axis does not affect any output axes, just drop input axis. If input axis affects only one output axis, drop both input and corresponding output. Similarly when specifying an output axis. If *axis_id* is a str, it must be unambiguous

- if the named axis exists in both input and output, and they do not correspond, raises a AxisError. See Raises section for checks

fix0: bool, optional
Whether to fix potential 0 TR in affine

Returns

cm_redux
[Affine] Affine coordinate map with orthogonal input + output dimension dropped

Raises

AxisError: if *axis_id* is a str and does not match any no input or output
coordinate names.

AxisError: if specified *axis_id* affects more than a single input / output
axis.

AxisError: if the named *axis_id* exists in both input and output, and they
do not correspond.

Examples

Typical use is in getting a 3D coordinate map from 4D

```
>>> cm4d = AffineTransform.from_params('ijkl', 'xyzt', np.diag([1,2,3,4,1]))
>>> cm3d = drop_io_dim(cm4d, 't')
>>> cm3d.affine
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  2.,  0.,  0.],
       [ 0.,  0.,  3.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

`nipy.core.reference.coordinate_map.`**`equivalent`**`(`*mapping1*, *mapping2*`)`

A test to see if mapping1 is equal to mapping2 after possibly reordering the domain and range of mapping.

Parameters

mapping1
[CoordinateMap or AffineTransform]

mapping2
[CoordinateMap or AffineTransform]

Returns

are_they_equal
[bool]

Examples

```
>>> ijk = CoordinateSystem('ijk')
>>> xyz = CoordinateSystem('xyz')
>>> T = np.random.standard_normal((4,4))
>>> T[-1] = [0,0,0,1] # otherwise AffineTransform raises
...                   # an exception because
...                   # it's supposed to represent an
...                   # affine transform in homogeneous
...                   # coordinates
>>> A = AffineTransform(ijk, xyz, T)
>>> B = A.reordered_domain('ikj').reordered_range('xzy')
>>> C = B.renamed_domain({'i':'slice'})
>>> equivalent(A, B)
True
>>> equivalent(A, C)
False
>>> equivalent(B, C)
False
>>>
>>> D = CoordinateMap(ijk, xyz, np.exp)
>>> equivalent(D, D)
True
>>> E = D.reordered_domain('kij').reordered_range('xzy')
>>> # no non-AffineTransform will ever be
>>> # equivalent to a reordered version of itself,
>>> # because their functions don't evaluate as equal
>>> equivalent(D, E)
False
>>> equivalent(E, E)
True
>>>
>>> # This has not changed the order
>>> # of the axes, so the function is still the same
>>>
>>> F = D.reordered_range('xyz').reordered_domain('ijk')
>>> equivalent(F, D)
True
>>> id(F) == id(D)
False
```

`nipy.core.reference.coordinate_map.`**`input_axis_index`**(*coordmap*, *axis_id*, *fix0=True*)

Return input axis index for *axis_id*

axis_id can be integer, or a name of an input axis, or it can be the name of an output axis which maps to an input axis.

Parameters

coordmap
[AffineTransform]

axis_id
[int or str] If int, then an index of an input axis. Can be negative, so that -2 refers to the second to last input axis. If a str can be the name of an input axis, or the name of an output

axis that should have a corresponding input axis (see Raises section).

fix0: bool, optional
Whether to fix potential single 0 on diagonal of affine. This often happens when loading nifti images with TR set to 0.

Returns

inax
[int] index of matching input axis. If *axis_id* is the name of an output axis, then *inax* will be the input axis that had a 'best' match with this output axis. The 'best' match algorithm ensures that there can only be one input axis paired with one output axis.

Raises

AxisError: if no matching name found
AxisError
[if name exists in both input and output and they do not map to] each other

AxisError
[if name present in output but no matching input]

`nipy.core.reference.coordinate_map.`**`io_axis_indices`**`(`*coordmap*, *axis_id*, *fix0=True*`)`

Return input and output axis index for id *axis_id* in *coordmap*

Parameters

cm
[class:*AffineTransform*] Affine coordinate map instance

axis_id
[int or str] If int, gives index of *input* axis. Can be negative, so that -2 refers to the second from last input axis. If str, gives name of input *or* output axis. If *axis_id* is a str, it must be unambiguous - if the named axis exists in both input and output, and they do not correspond, raises a AxisError. See Raises section for checks

fix0: bool, optional
Whether to fix potential 0 column / row in affine

Returns

in_index
[None or int] index of input axis that corresponds to *axis_id*

out_index
[None or int] index of output axis that corresponds to *axis_id*

Raises

AxisError: if *axis_id* is a str and does not match any input or output
coordinate names.

AxisError: if the named *axis_id* exists in both input and output, and they
do not correspond.

Examples

```
>>> aff = [[0, 1, 0, 10], [1, 0, 0, 11], [0, 0, 1, 12], [0, 0, 0, 1]]
>>> cmap = AffineTransform('ijk', 'xyz', aff)
>>> io_axis_indices(cmap, 0)
(0, 1)
>>> io_axis_indices(cmap, 1)
(1, 0)
>>> io_axis_indices(cmap, -1)
(2, 2)
>>> io_axis_indices(cmap, 'j')
(1, 0)
>>> io_axis_indices(cmap, 'y')
(0, 1)
```

`nipy.core.reference.coordinate_map.`**`orth_axes`**(*in_ax*, *out_ax*, *affine*, *allow_zero=True*, *tol=1e-05*)

True if *in_ax* related only to *out_ax* in *affine* and vice versa

Parameters

in_ax
[int] Input axis index

out_ax
[int] Output axis index

affine
[array-like] Affine transformation matrix

allow_zero
[bool, optional] Whether to allow zero in `affine[out_ax, in_ax]`. This means that the two axes are not related, but nor is this pair related to any other part of the affine.

Returns

tf
[bool] True if in_ax, out_ax pair are orthogonal to the rest of *affine*, unless *allow_zero* is False, in which case require in addition that `affine[out_ax, in_ax] != 0`.

Examples

```
>>> aff = np.eye(4)
>>> orth_axes(1, 1, aff)
True
>>> orth_axes(1, 2, aff)
False
```

`nipy.core.reference.coordinate_map.`**`product`**(**cmaps*, ***kwargs*)

"topological" product of two or more mappings

The mappings can be either AffineTransforms or CoordinateMaps.

If they are all AffineTransforms, the result is an AffineTransform, else it is a CoordinateMap.

Parameters

cmaps
[sequence of CoordinateMaps or AffineTransforms]

Returns

cmap
[CoordinateMap]

Examples

```
>>> inc1 = AffineTransform.from_params('i', 'x', np.diag([2,1]))
>>> inc2 = AffineTransform.from_params('j', 'y', np.diag([3,1]))
>>> inc3 = AffineTransform.from_params('k', 'z', np.diag([4,1]))
```

```
>>> cmap = product(inc1, inc3, inc2)
>>> cmap.function_domain.coord_names
('i', 'k', 'j')
>>> cmap.function_range.coord_names
('x', 'z', 'y')
>>> cmap.affine
array([[ 2.,  0.,  0.,  0.],
       [ 0.,  4.,  0.,  0.],
       [ 0.,  0.,  3.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

```
>>> A1 = AffineTransform.from_params('ij', 'xyz', np.array([[2,3,1,0],[3,4,5,0],[7,
↪9,3,1]]).T)
>>> A2 = AffineTransform.from_params('xyz', 'de', np.array([[8,6,7,4],[1,-1,13,3],
↪[0,0,0,1]]))
```

```
>>> A1.affine
array([[ 2.,  3.,  7.],
       [ 3.,  4.,  9.],
       [ 1.,  5.,  3.],
       [ 0.,  0.,  1.]])
>>> A2.affine
array([[  8.,   6.,   7.,   4.],
       [  1.,  -1.,  13.,   3.],
       [  0.,   0.,   0.,   1.]])
```

```
>>> p=product(A1, A2)
>>> p.affine
array([[  2.,   3.,   0.,   0.,   0.,   7.],
       [  3.,   4.,   0.,   0.,   0.,   9.],
       [  1.,   5.,   0.,   0.,   0.,   3.],
       [  0.,   0.,   8.,   6.,   7.,   4.],
       [  0.,   0.,   1.,  -1.,  13.,   3.],
       [  0.,   0.,   0.,   0.,   0.,   1.]])
```

```
>>> np.allclose(p.affine[:3,:2], A1.affine[:3,:2])
True
>>> np.allclose(p.affine[:3,-1], A1.affine[:3,-1])
True
>>> np.allclose(p.affine[3:5,2:5], A2.affine[:2,:3])
```

(continues on next page)

(continued from previous page)

```
True
>>> np.allclose(p.affine[3:5,-1], A2.affine[:2,-1])
True
>>>
```

```
>>> A1([3,4])
array([ 25.,  34.,  26.])
>>> A2([5,6,7])
array([ 129.,   93.])
>>> p([3,4,5,6,7])
array([  25.,   34.,   26.,  129.,   93.])
```

nipy.core.reference.coordinate_map.**renamed_domain**(*mapping*, *newnames*, *name=''*)

New coordmap with the coordinates of function_domain renamed

Parameters

newnames: dict
A dictionary whose keys are integers or are in mapping.function_range.coord_names and whose values are the new names.

Returns

newmapping
[CoordinateMap or AffineTransform] A new mapping with renamed function_domain. If isinstance(mapping, AffineTransform), newmapping is also an AffineTransform. Otherwise, it is a CoordinateMap.

Examples

```
>>> affine_domain = CoordinateSystem('ijk')
>>> affine_range = CoordinateSystem('xyz')
>>> affine_matrix = np.identity(4)
>>> affine_mapping = AffineTransform(affine_domain, affine_range, affine_matrix)
```

```
>>> new_affine_mapping = affine_mapping.renamed_domain({'i':'phase','k':'freq','j':
↪'slice'})
>>> new_affine_mapping.function_domain
CoordinateSystem(coord_names=('phase', 'slice', 'freq'), name='', coord_
↪dtype=float64)
```

```
>>> new_affine_mapping = affine_mapping.renamed_domain({'i':'phase','k':'freq','l':
↪'slice'})
Traceback (most recent call last):
   ...
ValueError: no domain coordinate named l
```

nipy.core.reference.coordinate_map.**renamed_range**(*mapping*, *newnames*)

New coordmap with the coordinates of function_range renamed

Parameters

newnames
[dict] A dictionary whose keys are integers or in mapping.function_range.coord_names and whose values are the new names.

Returns

newmapping
[CoordinateMap or AffineTransform] A new CoordinateMap with the coordinates of function_range renamed. If isinstance(mapping, AffineTransform), newmapping is also an AffineTransform. Otherwise, it is a CoordinateMap.

Examples

```
>>> affine_domain = CoordinateSystem('ijk')
>>> affine_range = CoordinateSystem('xyz')
>>> affine_matrix = np.identity(4)
>>> affine_mapping = AffineTransform(affine_domain, affine_range, affine_matrix)
>>> new_affine_mapping = affine_mapping.renamed_range({'x':'u'})
>>> new_affine_mapping.function_range
CoordinateSystem(coord_names=('u', 'y', 'z'), name='', coord_dtype=float64)
```

```
>>> new_affine_mapping = affine_mapping.renamed_range({'w':'u'})
Traceback (most recent call last):
   ...
ValueError: no range coordinate named w
```

nipy.core.reference.coordinate_map.**reordered_domain**(*mapping*, *order=None*)

New coordmap with the coordinates of function_domain reordered

Default behaviour is to reverse the order of the coordinates.

Parameters

order: sequence
Order to use, defaults to reverse. The elements can be integers, strings or 2-tuples of strings. If they are strings, they should be in mapping.function_domain.coord_names.

Returns

newmapping
[CoordinateMap or AffineTransform] A new CoordinateMap with the coordinates of function_domain reordered. If isinstance(mapping, AffineTransform), newmapping is also an AffineTransform. Otherwise, it is a CoordinateMap.

Notes

If no reordering is to be performed, it returns a copy of mapping.

Examples

```
>>> input_cs = CoordinateSystem('ijk')
>>> output_cs = CoordinateSystem('xyz')
>>> cm = AffineTransform(input_cs, output_cs, np.identity(4))
>>> cm.reordered_domain('ikj').function_domain
CoordinateSystem(coord_names=('i', 'k', 'j'), name='', coord_dtype=float64)
```

nipy.core.reference.coordinate_map.**reordered_range**(*mapping*, *order=None*)

New coordmap with the coordinates of function_range reordered

Defaults to reversing the coordinates of function_range.

Parameters

order: sequence
Order to use, defaults to reverse. The elements can be integers, strings or 2-tuples of strings. If they are strings, they should be in mapping.function_range.coord_names.

Returns

newmapping
[CoordinateMap or AffineTransform] A new CoordinateMap with the coordinates of function_range reordered. If isinstance(mapping, AffineTransform), newmapping is also an AffineTransform. Otherwise, it is a CoordinateMap.

Notes

If no reordering is to be performed, it returns a copy of mapping.

Examples

```
>>> input_cs = CoordinateSystem('ijk')
>>> output_cs = CoordinateSystem('xyz')
>>> cm = AffineTransform(input_cs, output_cs, np.identity(4))
>>> cm.reordered_range('xzy').function_range
CoordinateSystem(coord_names=('x', 'z', 'y'), name='', coord_dtype=float64)
>>> cm.reordered_range([0,2,1]).function_range.coord_names
('x', 'z', 'y')
```

```
>>> newcm = cm.reordered_range('yzx')
>>> newcm.function_range.coord_names
('y', 'z', 'x')
```

nipy.core.reference.coordinate_map.**shifted_domain_origin**(*mapping*, *difference_vector*, *new_origin*)

Shift the origin of the domain

Parameters

difference_vector
[array] Representing the difference shifted_origin-current_origin in the domain's basis.

Examples

```
>>> A = np.random.standard_normal((5,6))
>>> A[-1] = [0,0,0,0,0,1]
>>> affine_transform = AffineTransform(CS('ijklm', 'oldorigin'), CS('xyzt'), A)
>>> affine_transform.function_domain
CoordinateSystem(coord_names=('i', 'j', 'k', 'l', 'm'), name='oldorigin', coord_
↪dtype=float64)
```

A random change of origin

```
>>> difference = np.random.standard_normal(5)
```

The same affine transformation with a different origin for its domain

```
>>> shifted_affine_transform = shifted_domain_origin(affine_transform, difference,
↪'neworigin')
>>> shifted_affine_transform.function_domain
CoordinateSystem(coord_names=('i', 'j', 'k', 'l', 'm'), name='neworigin', coord_
↪dtype=float64)
```

Let's check that things work

```
>>> point_in_old_basis = np.random.standard_normal(5)
```

This is the relation ship between coordinates in old and new origins

```
>>> np.allclose(shifted_affine_transform(point_in_old_basis), affine_
↪transform(point_in_old_basis+difference))
True
>>> np.allclose(shifted_affine_transform(point_in_old_basis-difference), affine_
↪transform(point_in_old_basis))
True
```

`nipy.core.reference.coordinate_map.`**`shifted_range_origin`**(*mapping*, *difference_vector*, *new_origin*)

Shift the origin of the range.

Parameters

difference_vector
[array] Representing the difference shifted_origin-current_origin in the range's basis.

Examples

```
>>> A = np.random.standard_normal((5,6))
>>> A[-1] = [0,0,0,0,0,1]
>>> affine_transform = AffineTransform(CS('ijklm'), CS('xyzt', 'oldorigin'), A)
>>> affine_transform.function_range
CoordinateSystem(coord_names=('x', 'y', 'z', 't'), name='oldorigin', coord_
↪dtype=float64)
```

Make a random shift of the origin in the range

```
>>> difference = np.random.standard_normal(4)
>>> shifted_affine_transform = shifted_range_origin(affine_transform, difference,
↪'neworigin')
>>> shifted_affine_transform.function_range
CoordinateSystem(coord_names=('x', 'y', 'z', 't'), name='neworigin', coord_
↪dtype=float64)
>>>
```

Evaluate the transform and verify it does as expected

```
>>> point_in_domain = np.random.standard_normal(5)
```

Check that things work

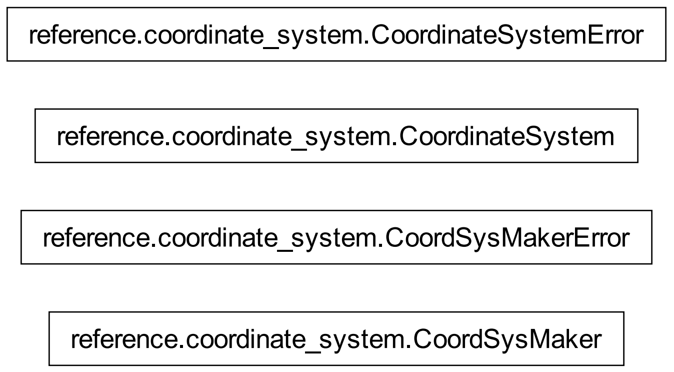
```
>>> np.allclose(shifted_affine_transform(point_in_domain), affine_transform(point_
↪in_domain) - difference)
True
>>> np.allclose(shifted_affine_transform(point_in_domain) + difference, affine_
↪transform(point_in_domain))
True
```

CHAPTER EIGHTYFIVE

CORE.REFERENCE.COORDINATE_SYSTEM

85.1 Module: `core.reference.coordinate_system`

Inheritance diagram for `nipy.core.reference.coordinate_system`:


reference.coordinate_system.CoordinateSystemError
reference.coordinate_system.CoordinateSystem
reference.coordinate_system.CoordSysMakerError
reference.coordinate_system.CoordSysMaker

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.

85.2 Classes

85.2.1 `CoordSysMaker`

class `nipy.core.reference.coordinate_system.`**`CoordSysMaker`**(*coord_names*, *name=''*, *coord_dtype=<class 'numpy.float64'>*)

Bases: `object`

Class to create similar coordinate maps of different dimensions

`__init__`(*coord_names*, *name=''*, *coord_dtype=<class 'numpy.float64'>*)

Create a coordsys maker with given axis *coord_names*

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.float64). The value can by anything that can be passed to the np.dtype constructor. For example `np.float64`, `np.dtype(np.float64)` or `f8` all result in the same `coord_dtype`.

Examples

```
>>> cmkr = CoordSysMaker('ijk', 'a name')
>>> print(cmkr(2))
CoordinateSystem(coord_names=('i', 'j'), name='a name', coord_dtype=float64)
>>> print(cmkr(3))
CoordinateSystem(coord_names=('i', 'j', 'k'), name='a name', coord_
↪dtype=float64)
```

`coord_sys_klass`

alias of *`CoordinateSystem`*

85.2.2 `CoordSysMakerError`

class `nipy.core.reference.coordinate_system.`**`CoordSysMakerError`**

Bases: `Exception`

`__init__`(**args*, ***kwargs*)

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

85.2.3 `CoordinateSystem`

class `nipy.core.reference.coordinate_system.`**`CoordinateSystem`**(*coord_names*, *name=''*, *coord_dtype=<class 'numpy.float64'>*)

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.float64)
>>> 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.float64) 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 and the same names and the same name.

```
>>> another_cs = CoordinateSystem(names, 'not irrelevant', np.float64)
>>> cs == another_cs
False
>>> cs.dtype == another_cs.dtype
True
>>> cs.name == another_cs.name
False
```

`__init__`(*coord_names*, *name=''*, *coord_dtype=<class 'numpy.float64'>*)

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.float64). The value can by anything that can be passed to the np.dtype constructor. For example `np.float64`, `np.dtype(np.float64)` or `f8` all result in the same `coord_dtype`.

Examples

```
>>> c = CoordinateSystem('ij', name='input')
>>> print(c)
CoordinateSystem(coord_names=('i', 'j'), name='input', coord_dtype=float64)
>>> c.coord_dtype
dtype('float64')
```

coord_dtype

alias of float64

coord_names = ('x', 'y', 'z')

dtype = dtype([('x', '<f8'), ('y', '<f8'), ('z', '<f8')])

index(*coord_name*)

Return the index of a given named coordinate.

```
>>> c = CoordinateSystem('ij', name='input')
>>> c.index('i')
0
>>> c.index('j')
1
```

name = 'world-LPI'

ndim = 3

similar_to(*other*)

Similarity is defined by self.dtype, ignoring name

Parameters

other
[*CoordinateSystem*] The object to be compared with

Returns

tf: bool

85.2.4 CoordinateSystemError

class nipy.core.reference.coordinate_system.**CoordinateSystemError**

Bases: Exception

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

85.3 Functions

nipy.core.reference.coordinate_system.**is_coordsys**(*obj*)

Test if *obj* has the CoordinateSystem API

Parameters

obj
[object] Object to test

Returns

tf
[bool] True if *obj* has the coordinate system API

Examples

```
>>> is_coordsys(CoordinateSystem('xyz'))
True
>>> is_coordsys(CoordSysMaker('ikj'))
False
```

nipy.core.reference.coordinate_system.**is_coordsys_maker**(*obj*)

Test if *obj* has the CoordSysMaker API

Parameters

obj
[object] Object to test

Returns

tf
[bool] True if *obj* has the coordinate system API

Examples

```
>>> is_coordsys_maker(CoordSysMaker('ikj'))
True
>>> is_coordsys_maker(CoordinateSystem('xyz'))
False
```

nipy.core.reference.coordinate_system.**product**(**coord_systems*, ***kwargs*)

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`]

name
[str] Name of output coordinate system

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))
CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name='product', coord_
↪dtype=complex128)
```

```
>>> print(product(c1, c2, name='another name'))
CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name='another name', coord_
↪dtype=complex128)
```

```
>>> product(c2, c3)
Traceback (most recent call last):
   ...
ValueError: coord_names must have distinct names
```

`nipy.core.reference.coordinate_system.`**`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.

Parameters

dtypes
[sequence of `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'))
Traceback (most recent call last):
...
TypeError: dtype must be valid numpy dtype int, uint, float, complex or object
```

```
>>> # 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)
Traceback (most recent call last):
...
TypeError: dtype must be valid numpy dtype int, uint, float, complex or object
```

CHAPTER

EIGHTYSIX

CORE.REFERENCE.SLICES

86.1 Module: `core.reference.slices`

A set of methods to get coordinate maps which represent slices in space.

86.2 Functions

`nipy.core.reference.slices.`**`bounding_box`**`(`*coordmap*, *shape*`)`

Determine a valid bounding box from a CoordinateMap and a shape.

Parameters

coordmap
[CoordinateMap or AffineTransform] Containing mapping between voxel coordinates implied by *shape* and physical coordinates.

shape
[sequence of int] shape implying array

Returns

limits
[(N,) tuple of (2,) tuples of float] minimum and maximum coordinate values in output space (range) of *coordmap*. N is given by coordmap.ndim[1].

Examples

Make a 3D voxel to mni coordmap

```
>>> from nipy.core.api import vox2mni
>>> affine = np.array([[1, 0, 0, 2],
...                    [0, 3, 0, 4],
...                    [0, 0, 5, 6],
...                    [0, 0, 0, 1]], dtype=np.float64)
>>> A = vox2mni(affine)
>>> bounding_box(A, (30,40,20))
((2.0, 31.0), (4.0, 121.0), (6.0, 101.0))
```

`nipy.core.reference.slices.`**`xslice`**`(`*x*, *y_spec*, *z_spec*, *world*`)`

Return an LPS slice through a 3d box with x fixed.

Parameters

x
[float] The value at which x is fixed.

y_spec
[sequence] A sequence with 2 values of form ((float, float), int). The (float, float) components are the min and max y values; the int is the number of points.

z_spec
[sequence] As for *y_spec* but for z

world
[str or CoordinateSystem CoordSysMaker or XYZSpace] World 3D space to which resulting coordmap refers

Returns

affine_transform
[AffineTransform] An affine transform that describes an plane in LPS coordinates with x fixed.

Examples

```
>>> y_spec = ([-114,114], 115) # voxels of size 2 in y, starting at -114, ending at␣
↪114
>>> z_spec = ([-70,100], 86) # voxels of size 2 in z, starting at -70, ending at 100
>>> x30 = xslice(30, y_spec, z_spec, 'scanner')
>>> x30([0,0])
array([  30., -114.,  -70.])
>>> x30([114,85])
array([  30.,  114.,  100.])
>>> x30
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i_y', 'i_z'), name='slice', coord_
↪dtype=float64),
   function_range=CoordinateSystem(coord_names=('scanner-x=L->R', 'scanner-y=P->A',
↪'scanner-z=I->S'), name='scanner', coord_dtype=float64),
   affine=array([[   0.,    0.,   30.],
                 [   2.,    0., -114.],
                 [   0.,    2.,  -70.],
                 [   0.,    0.,    1.]])
)
>>> bounding_box(x30, (y_spec[1], z_spec[1]))
((30.0, 30.0), (-114.0, 114.0), (-70.0, 100.0))
```

`nipy.core.reference.slices.`**`yslice`**(*y*, *x_spec*, *z_spec*, *world*)

Return a slice through a 3d box with y fixed.

Parameters

y
[float] The value at which y is fixed.

x_spec
[sequence] A sequence with 2 values of form ((float, float), int). The (float, float) components are the min and max x values; the int is the number of points.

z_spec
[sequence] As for *x_spec* but for z

world
[str or CoordinateSystem CoordSysMaker or XYZSpace] World 3D space to which resulting coordmap refers

Returns

affine_transform
[AffineTransform] An affine transform that describes an plane in LPS coordinates with y fixed.

Examples

```
>>> x_spec = ([-92,92], 93) # voxels of size 2 in x, starting at -92, ending at 92
>>> z_spec = ([-70,100], 86) # voxels of size 2 in z, starting at -70, ending at 100
>>> y70 = yslice(70, x_spec, z_spec, 'mni')
>>> y70
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i_x', 'i_z'), name='slice', coord_
↪dtype=float64),
   function_range=CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-
↪z=I->S'), name='mni', coord_dtype=float64),
   affine=array([[  2.,   0., -92.],
                 [  0.,   0.,  70.],
                 [  0.,   2., -70.],
                 [  0.,   0.,   1.]])
)
>>> y70([0,0])
array([-92.,  70., -70.])
>>> y70([92,85])
array([  92.,   70.,  100.])
>>> bounding_box(y70, (x_spec[1], z_spec[1]))
((-92.0, 92.0), (70.0, 70.0), (-70.0, 100.0))
```

`nipy.core.reference.slices.`**`zslice`**(*z*, *x_spec*, *y_spec*, *world*)

Return a slice through a 3d box with z fixed.

Parameters

z
[float] The value at which z is fixed.

x_spec
[sequence] A sequence with 2 values of form ((float, float), int). The (float, float) components are the min and max x values; the int is the number of points.

y_spec
[sequence] As for *x_spec* but for y

world
[str or CoordinateSystem CoordSysMaker or XYZSpace] World 3D space to which resulting coordmap refers

Returns

affine_transform
[AffineTransform] An affine transform that describes a plane in LPS coordinates with z fixed.

Examples

```
>>> x_spec = ([-92,92], 93) # voxels of size 2 in x, starting at -92, ending at 92
>>> y_spec = ([-114,114], 115) # voxels of size 2 in y, starting at -114, ending at␣
↪114
>>> z40 = zslice(40, x_spec, y_spec, 'unknown')
>>> z40
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i_x', 'i_y'), name='slice', coord_
↪dtype=float64),
   function_range=CoordinateSystem(coord_names=('unknown-x=L->R', 'unknown-y=P->A',
↪'unknown-z=I->S'), name='unknown', coord_dtype=float64),
   affine=array([[   2.,    0.,  -92.],
                 [   0.,    2., -114.],
                 [   0.,    0.,   40.],
                 [   0.,    0.,    1.]])
)
>>> z40([0,0])
array([ -92., -114.,   40.])
>>> z40([92,114])
array([  92.,  114.,   40.])
>>> bounding_box(z40, (x_spec[1], y_spec[1]))
((-92.0, 92.0), (-114.0, 114.0), (40.0, 40.0))
```

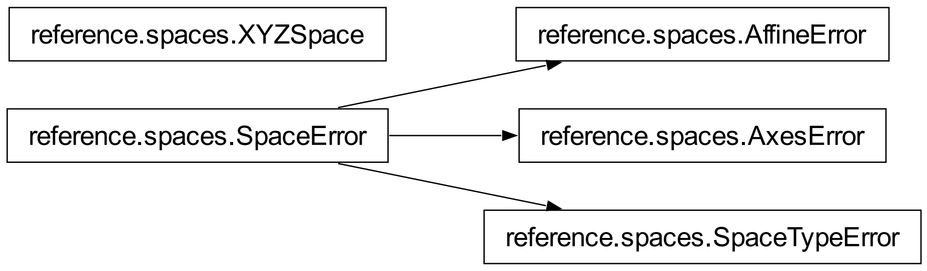
CHAPTER

EIGHTYSEVEN

CORE.REFERENCE.SPACES

87.1 Module: `core.reference.spaces`

Inheritance diagram for `nipy.core.reference.spaces`:


reference.spaces.XYZSpace
reference.spaces.AffineError
reference.spaces.SpaceError
reference.spaces.AxesError
reference.spaces.SpaceTypeError

Useful neuroimaging coordinate map makers and utilities

87.2 Classes

87.2.1 `AffineError`

class nipy.core.reference.spaces.**AffineError**

Bases: *SpaceError*

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

87.2.2 AxesError

class nipy.core.reference.spaces.**AxesError**

Bases: *SpaceError*

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

87.2.3 SpaceError

class nipy.core.reference.spaces.**SpaceError**

Bases: Exception

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

87.2.4 SpaceTypeError

class nipy.core.reference.spaces.**SpaceTypeError**

Bases: *SpaceError*

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

87.2.5 XYZSpace

class nipy.core.reference.spaces.**XYZSpace**(*name*)

Bases: object

Class contains logic for spaces with XYZ coordinate systems

```
>>> sp = XYZSpace('hijo')
>>> print(sp)
hijo: [('x', 'hijo-x=L->R'), ('y', 'hijo-y=P->A'), ('z', 'hijo-z=I->S')]
>>> csm = sp.to_coordsys_maker()
>>> cs = csm(3)
>>> cs
CoordinateSystem(coord_names=('hijo-x=L->R', 'hijo-y=P->A', 'hijo-z=I->S'), name=
↪'hijo', coord_dtype=float64)
>>> cs in sp
True
```

`__init__`(*name*)

`as_map`()

Return xyz names as dictionary

```
>>> sp = XYZSpace('hijo')
>>> sorted(sp.as_map().items())
[('x', 'hijo-x=L->R'), ('y', 'hijo-y=P->A'), ('z', 'hijo-z=I->S')]
```

`as_tuple`()

Return xyz names as tuple

```
>>> sp = XYZSpace('hijo')
>>> sp.as_tuple()
('hijo-x=L->R', 'hijo-y=P->A', 'hijo-z=I->S')
```

`register_to`(*mapping*)

Update *mapping* with key=self.x, value='x' etc pairs

The mapping will then have keys that are names we (`self`) identify as being x, or y, or z, values are 'x' or 'y' or 'z'.

Note that this is the opposite way round for keys, values, compared to the `as_map` method.

Parameters

mapping
[mapping] such as a dict

Returns

None

Examples

```
>>> sp = XYZSpace('hijo')
>>> mapping = {}
>>> sp.register_to(mapping)
>>> sorted(mapping.items())
[('hijo-x=L->R', 'x'), ('hijo-y=P->A', 'y'), ('hijo-z=I->S', 'z')]
```

`to_coordsys_maker`(*extras=()*)

Make a coordinate system maker for this space

Parameters

extra
[sequence] names for any further axes after x, y, z

Returns

csm
[CoordinateSystemMaker]

Examples

```
>>> sp = XYZSpace('hijo')
>>> csm = sp.to_coordsys_maker()
>>> csm(3)
CoordinateSystem(coord_names=('hijo-x=L->R', 'hijo-y=P->A', 'hijo-z=I->S'),␣
↪name='hijo', coord_dtype=float64)
```

`property x`

x-space coordinate name

`x_suffix = 'x=L->R'`

`property y`

y-space coordinate name

`y_suffix = 'y=P->A'`

`property z`

z-space coordinate name

`z_suffix = 'z=I->S'`

87.3 Functions

`nipy.core.reference.spaces.`**`get_world_cs`**(*world_id*, *ndim=3*, *extras='tuvw'*, *spaces=None*)

Get world coordinate system from *world_id*

Parameters

world_id
[str, XYZSPace, CoordSysMaker or CoordinateSystem] Object defining a world output system. If str, then should be a name of an XYZSpace in the list *spaces*.

ndim
[int, optional] Number of dimensions in this world. Default is 3

extras
[sequence, optional] Coordinate (axis) names for axes > 3 that are not named by *world_id*

spaces
[None or sequence, optional] List of known (named) spaces to compare a str *world_id* to. If None, use the module level `known_spaces`

Returns

world_cs
[CoordinateSystem] A world coordinate system

Examples

```
>>> get_world_cs('mni')
CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-z=I->S'), name='mni',
↪ coord_dtype=float64)
```

```
>>> get_world_cs(mni_space, 4)
CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-z=I->S', 't'), name=
↪'mni', coord_dtype=float64)
```

```
>>> from nipy.core.api import CoordinateSystem
>>> get_world_cs(CoordinateSystem('xyz'))
CoordinateSystem(coord_names=('x', 'y', 'z'), name='', coord_dtype=float64)
```

nipy.core.reference.spaces.**is_xyz_affable**(*coordmap*, *name2xyz=None*)

Return True if the coordap has an xyz affine

Parameters

coordmap
[CoordinateMap instance] Coordinate map to test

name2xyz
[None or mapping, optional] Object such that name2xyz[ax_name] returns 'x', or 'y' or 'z' or raises a KeyError for a str ax_name. None means use module default.

Returns

tf
[bool] True if *coordmap* has an xyz affine, False otherwise

Examples

```
>>> cmap = vox2mni(np.diag([2,3,4,5,1]))
>>> cmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name='voxels',
↪ coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-
↪z=I->S', 't'), name='mni', coord_dtype=float64),
   affine=array([[ 2.,  0.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.,  0.],
                 [ 0.,  0.,  4.,  0.,  0.],
                 [ 0.,  0.,  0.,  5.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
>>> is_xyz_affable(cmap)
True
>>> time0_cmap = cmap.reordered_domain([3,0,1,2])
>>> time0_cmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('l', 'i', 'j', 'k'), name='voxels',
↪ coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-
```

(continues on next page)

(continued from previous page)

```
↪z=I->S', 't'), name='mni', coord_dtype=float64),
   affine=array([[ 0.,  2.,  0.,  0.,  0.],
                 [ 0.,  0.,  3.,  0.,  0.],
                 [ 0.,  0.,  0.,  4.,  0.],
                 [ 5.,  0.,  0.,  0.,  0.],
                 [ 0.,  0.,  0.,  0.,  1.]])
)
>>> is_xyz_affable(time0_cmap)
False
```

nipy.core.reference.spaces.**is_xyz_space**(*obj*)

True if *obj* appears to be an XYZ space definition

nipy.core.reference.spaces.**known_space**(*obj*, *spaces=None*)

If *obj* is in a known space, return the space, otherwise return None

Parameters

obj
[object] Object that can be tested against an XYZSpace with `obj in sp`

spaces
[None or sequence, optional] spaces to test against. If None, use the module level `known_spaces` list to test against.

Returns

sp
[None or XYZSpace] If *obj* is not in any of the *known_spaces*, return None. Otherwise return the first matching space in *known_spaces*

Examples

```
>>> from nipy.core.api import CoordinateSystem
>>> sp0 = XYZSpace('hijo')
>>> sp1 = XYZSpace('hija')
```

Make a matching coordinate system

```
>>> cs = sp0.to_coordsys_maker()(3)
```

Test whether this coordinate system is in either of `(sp0, sp1)`

```
>>> known_space(cs, (sp0, sp1))
XYZSpace('hijo')
```

So, yes, it's in `sp0`. How about another generic CoordinateSystem?

```
>>> known_space(CoordinateSystem('xyz'), (sp0, sp1)) is None
True
```

So, no, that is not in either of `(sp0, sp1)` nipy.core.reference.spaces.**xyz_affine**(*coordmap*, *name2xyz=None*)

Return (4, 4) affine mapping voxel coordinates to XYZ from *coordmap*

If no (4, 4) affine "makes sense"(TM) for this *coordmap* then raise errors listed below. A (4, 4) affine makes sense if the first three output axes are recognizably X, Y, and Z in that order AND they there are corresponding input dimensions, AND the corresponding input dimensions are the first three input dimension (in any order). Thus the input axes have to be 3D.

Parameters

coordmap
[CoordinateMap instance]

name2xyz
[None or mapping, optional] Object such that name2xyz[ax_name] returns 'x', or 'y' or 'z' or raises a KeyError for a str ax_name. None means use module default.

Returns

xyz_aff
[(4,4) array] voxel to X, Y, Z affine mapping

Raises

SpaceTypeError
[if this is not an affine coordinate map]

AxesError
[if not all of x, y, z recognized in *coordmap* output, or they]

are in the wrong order, or the x, y, z axes do not correspond to the first three input axes.
AffineError
[if axes dropped from the affine contribute to x, y, z]

coordinates.

Notes

We could also try and "make sense" (TM) of a coordmap that had X, Y and Z outputs, but not in that order, nor all in the first three axes. In that case we could just permute the affine to get the output order we need. But, that could become confusing if the returned affine has different output coordinates than the passed *coordmap*. And it's more complicated. So, let's not do that for now.

Examples

```
>>> cmap = vox2mni(np.diag([2,3,4,5,1]))
>>> cmap
AffineTransform(
   function_domain=CoordinateSystem(coord_names=('i', 'j', 'k', 'l'), name='voxels',
↪ coord_dtype=float64),
   function_range=CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-
↪z=I->S', 't'), name='mni', coord_dtype=float64),
   affine=array([[ 2.,  0.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.,  0.],
                 [ 0.,  0.,  4.,  0.,  0.],
                 [ 0.,  0.,  0.,  5.,  0.],
```

(continues on next page)

(continued from previous page)

```
                 [ 0.,  0.,  0.,  0.,  1.]])
)
>>> xyz_affine(cmap)
array([[ 2.,  0.,  0.,  0.],
       [ 0.,  3.,  0.,  0.],
       [ 0.,  0.,  4.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

nipy.core.reference.spaces.**xyz_order**(*coordsys*, *name2xyz=None*)

Vector of orders for sorting coordsys axes in xyz first order

Parameters

coordsys
[CoordinateSystem instance]

name2xyz
[None or mapping, optional] Object such that name2xyz[ax_name] returns 'x', or 'y' or 'z' or raises a KeyError for a str ax_name. None means use module default.

Returns

xyz_order
[list] Ordering of axes to get xyz first ordering. See the examples.

Raises

AxesError
[if there are not all of x, y and z axes]

Examples

```
>>> from nipy.core.api import CoordinateSystem
>>> xyzt_cs = mni_csm(4) # coordsys with t (time) last
>>> xyzt_cs
CoordinateSystem(coord_names=('mni-x=L->R', 'mni-y=P->A', 'mni-z=I->S', 't'), name=
↪'mni', coord_dtype=float64)
>>> xyz_order(xyzt_cs)
[0, 1, 2, 3]
>>> tzyx_cs = CoordinateSystem(xyzt_cs.coord_names[::-1], 'reversed')
>>> tzyx_cs
CoordinateSystem(coord_names=('t', 'mni-z=I->S', 'mni-y=P->A', 'mni-x=L->R'), name=
↪'reversed', coord_dtype=float64)
>>> xyz_order(tzyx_cs)
[3, 2, 1, 0]
```

CHAPTER

EIGHTYEIGHT

CORE.UTILS.GENERATORS

88.1 Module: `core.utils.generators`

This module defines a few common generators for slicing over arrays.

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

88.2 Functions

`nipy.core.utils.generators.`**`data_generator`**(*data*, *iterable=None*)

Return generator for `[(i, data[i]) for i in iterable]`

If iterable is None, it defaults to range(data.shape[0])

Examples

```
>>> a = np.asarray([[True,False],[False,True]])
>>> b = np.asarray([[False,False],[True,False]])
```

```
>>> for i, d in data_generator(np.asarray([[1,2],[3,4]]), [a,b]):
...     print(d)
...
[1 4]
[3]
```

`nipy.core.utils.generators.`**`f_generator`**(*f*, *iterable*)

Return a generator for `[(i, f(x)) for i, x in iterable]`

Examples

```
>>> 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]
```

`nipy.core.utils.generators.`**`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:]))

`nipy.core.utils.generators.`**`parcels`**(*data*, *labels=None*, *exclude=()*)

Return a generator for `[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]
```

Parameters

data
[image or array-like] Either an image (with `get_fdata` method returning ndarray) or an array-like

labels
[iterable, optional] A sequence of labels for which to return indices within *data*. The elements in *labels* can themselves be lists, tuples, in which case the indices returned are for all values in *data* matching any of the items in the list, tuple.

exclude
[iterable, optional] Values in *labels* for which you do not want to return a parcel.

Returns

gen
[generator] generator yielding a array of boolean indices into *data* for which `data == label`, for each element in *label*.

Examples

```
>>> 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]]
```

(continues on next page)

(continued from previous page)

```
>>> for p in parcels([[1,1],[2,3]], labels=[(2,3),2]):
...     print(p)
...
[[False False]
 [ True  True]]
[[False False]
 [ True False]]
```

nipy.core.utils.generators.**shape_generator**(*img*, *shape*)

From a generator of items (i, r), return (i, r.reshape(shape))

nipy.core.utils.generators.**slice_generator**(*data*, *axis=0*)

Return generator for yielding slices along *axis*

Parameters

data
[array-like]

axis
[int or list or tuple] If int, gives the axis. If list or tuple, gives the combination of axes over which to iterate. First axis is fastest changing in output.

Examples

```
>>> 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]
```

nipy.core.utils.generators.**slice_parcels**(*data*, *labels=None*, *axis=0*)

A generator for slicing through parcels and slices of data…

hmmm… a better description is needed

```
>>> x=np.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):
```

(continues on next page)

(continued from previous page)

```
...     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]
```

nipy.core.utils.generators.**write_data**(*output*, *iterable*)

Write (index, data) iterable to *output*

Write some data to *output*. Iterable should return 2-tuples of the form index, data such that:

```
output[index] = data
```

makes sense.

Examples

```
>>> a=np.zeros((2,2))
>>> write_data(a, data_generator(np.asarray([[1,2],[3,4]])))
>>> a
array([[ 1.,  2.],
       [ 3.,  4.]])
```

CHAPTER

EIGHTYNINE

INTERFACES.MATLAB

89.1 Module: `interfaces.matlab`

General matlab interface code

This is for nipy convenience. If you're doing heavy matlab interfacing, please use NiPype instead:

http://nipy.org/nipype

89.2 Functions

`nipy.interfaces.matlab.`**`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

[file-like object]

Examples

```
>>> f = mlab_tempfile()
>>> pth, fname = os.path.split(f.name)
>>> '-' not in fname
True
>>> f.close()
```

`nipy.interfaces.matlab.`**`run_matlab`**`(`*cmd*`)`

`nipy.interfaces.matlab.`**`run_matlab_script`**`(`*script_lines*, *script_name='pyscript'*`)`

Put multiline matlab script into script file and run

CHAPTER

NINETY

INTERFACES.SPM

90.1 Module: `interfaces.spm`

Inheritance diagram for `nipy.interfaces.spm`:

Interfaces to SPM

90.2 Class

90.3 `SpmInfo`

class `nipy.interfaces.spm.`**`SpmInfo`**

Bases: `object`

`__init__`()

property spm_path

property spm_ver

90.4 Functions

`nipy.interfaces.spm.`**`fltcols`**(*vals*)

Trivial little function to make 1xN float vector

`nipy.interfaces.spm.`**`fname_presuffix`**(*fname*, *prefix=''*, *suffix=''*, *use_ext=True*)

`nipy.interfaces.spm.`**`fnames_presuffix`**(*fnames*, *prefix=''*, *suffix=''*)

`nipy.interfaces.spm.`**`make_job`**(*jobtype*, *jobname*, *contents*) `nipy.interfaces.spm.`**`run_jobdef`**`(`*jobdef*`)`

`nipy.interfaces.spm.`**`scans_for_fname`**`(`*fname*`)`

`nipy.interfaces.spm.`**`scans_for_fnames`**`(`*fnames*`)`

CHAPTER

NINETYONE

IO.FILES

91.1 Module: `io.files`

The io.files module provides basic functions for working with file-based images in nipy.

- load : load an image from a file
- save : save an image to a file

91.1.1 Examples

See documentation for load and save functions for worked examples.

91.2 Functions

`nipy.io.files.as_image`(*image_input*)

Load image from filename or pass through image instance

Parameters

image_input
[str or Image instance] image or string filename of image. If a string, load image and return. If an image, pass through without modification

Returns

img
[Image or Image-like instance] Input object if *image_input* seemed to be an image, loaded Image object if *image_input* was a string.

Raises

TypeError
[if neither string nor image-like passed]

Examples

```
>>> from nipy.testing import anatfile
>>> from nipy.io.api import load_image
>>> img = as_image(anatfile)
>>> img2 = as_image(img)
>>> img2 is img
True
```

`nipy.io.files.`**`load`**(*filename*)

Load an image from the given filename.

Parameters

filename
[string] Should resolve to a complete filename path.

Returns

image
[An *Image* object] If successful, a new *Image* object is returned.

See also:

`save_image`
function for saving images

`Image`
image object

Examples

```
>>> from nipy.io.api import load_image
>>> from nipy.testing import anatfile
>>> img = load_image(anatfile)
>>> img.shape
(33, 41, 25)
```

`nipy.io.files.`**`save`**(*img*, *filename*, *dtype_from='data'*)

Write the image to a file.

Parameters

img
[An *Image* object]

filename
[string] Should be a valid filename.

dtype_from
[{'data', 'header'} or dtype specifier, optional] Method of setting dtype to save data to disk. Value of 'data' (default), means use data dtype to save. 'header' means use data dtype specified in header, if available, otherwise use data dtype. Can also be any valid specifier for a numpy dtype, e.g. 'i4', `np.float32`. Not every format supports every dtype, so some values of this parameter or data dtypes will raise errors.

Returns

image
[An *Image* object] Possibly modified by saving.

See also:

`load_image`
function for loading images

`Image`
image object

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']
- SPM Analyze : ['.img', '.img.gz']

Examples

Make a temporary directory to store files

```
>>> import os
>>> from tempfile import mkdtemp
>>> tmpdir = mkdtemp()
```

Make some some files and save them

```
>>> import numpy as np
>>> from nipy.core.api import Image, AffineTransform
>>> from nipy.io.api import save_image
>>> data = np.zeros((91,109,91), dtype=np.uint8)
>>> cmap = AffineTransform('kji', 'zxy', np.eye(4))
>>> img = Image(data, cmap)
>>> fname1 = os.path.join(tmpdir, 'img1.nii.gz')
>>> saved_img1 = save_image(img, fname1)
>>> saved_img1.shape
(91, 109, 91)
>>> fname2 = os.path.join(tmpdir, 'img2.img.gz')
>>> saved_img2 = save_image(img, fname2)
>>> saved_img2.shape
(91, 109, 91)
>>> fname = 'test.mnc'
>>> saved_image3 = save_image(img, fname)
Traceback (most recent call last):
   ...
ValueError: Sorry, we cannot yet save as format "minc"
```

Finally, we clear up our temporary files:

```
>>> import shutil
>>> shutil.rmtree(tmpdir)
```

CHAPTER NINETYTWO

IO.NIBCOMPAT

92.1 Module: `io.nibcompat`

Compatibility functions for older versions of nibabel

Nibabel <= 1.3.0 do not have these attributes:

- header
- affine
- dataobj

The equivalents for these older versions of nibabel are:

- obj.get_header()
- obj.get_affine()
- obj._data

With old nibabel, getting unscaled data used *read_img_data(img, prefer="unscaled"). Newer nibabel should prefer the `get_unscaled* method on the image proxy object

92.2 Functions

`nipy.io.nibcompat.`**`get_affine`**`(`*img*`)`

Return affine from nibabel image

Parameters

img
[`SpatialImage` instance] Instance of nibabel `SpatialImage` class

Returns

affine
[object] affine object from *img*

`nipy.io.nibcompat.`**`get_dataobj`**`(`*img*`)`

Return data object for nibabel image

Parameters

img
[`SpatialImage` instance] Instance of nibabel `SpatialImage` class

Returns

dataobj
[object] `ArrayProxy` or ndarray object containing data for *img*

`nipy.io.nibcompat.`**`get_header`**`(`*img*`)`

Return header from nibabel image

Parameters

img
[`SpatialImage` instance] Instance of nibabel `SpatialImage` class

Returns

header
[object] header object from *img*

`nipy.io.nibcompat.`**`get_unscaled_data`**`(`*img*`)`

Get the data from a nibabel image, maybe without applying scaling

Parameters

img
[`SpatialImage` instance] Instance of nibabel `SpatialImage` class

Returns

data
[ndarray] Data as loaded from image, not applying scaling if this can be avoided

CHAPTER

NINETYTHREE

IO.NIFTI_REF

93.1 Module: `io.nifti_ref`

Inheritance diagram for `nipy.io.nifti_ref`:

An implementation of some of the NIFTI conventions as described in:

http://nifti.nimh.nih.gov/pub/dist/src/niftilib/nifti1.h

A version of the same file is in the nibabel repisitory at `doc/source/external/nifti1.h`.

93.1.1 Background

We (nipystas) make an explicit distinction between:

- an input coordinate system of an image (the array == voxel coordinates)
- output coordinate system (usually millimeters in some world for space, seconds for time)
- the mapping between the two.

The collection of these three is the `coordmap` attribute of a NIPY image.

There is no constraint that the number of input and output coordinates should be the same.

We don't specify the units of our output coordinate system, but assume spatial units are millimeters and time units are seconds.

NIFTI is mostly less explicit, but more constrained.

NIFTI input coordinate system

NIFTI files can have up to seven voxel dimensions (7 axes in the input coordinate system).

The first 3 voxel dimensions of a NIFTI file must be spatial but can be in any order in relationship to directions in mm space (the output coordinate system)

The 4th voxel dimension is assumed to be time. In particular, if you have some other meaning for a non-spatial dimension, the NIFTI standard suggests you set the length of the 4th dimension to be 1, and use the 5th dimension of the image instead, and set the NIFTI "intent" fields to state the meaning. If the `intent` field is set correctly then it should be possible to set meaningful input coordinate axis names for dimensions > (0, 1, 2).

There's a wrinkle to the 4th axis is time story; the `xyxt_units` field in the NIFTI header can specify the 4th dimension units as Hz (frequency), PPM (concentration) or Radians / second.

NIFTI also has a 'dim_info' header attribute that optionally specifies that 0 or more of the first three voxel axes are 'frequency', 'phase' or 'slice'. These terms refer to 2D MRI acquisition encoding, where 'slice's are collected sequentially, and the two remaining dimensions arose from frequency and phase encoding. The `dim_info` fields are often not set. 3D acquisitions don't have a 'slice' dimension.

NIFTI output coordinate system

In the NIFTI specification, the order of the output coordinates (at least the first 3) are fixed to be what might be called RAS+, that is ('x=L->R', 'y=P->A', 'z=I->S'). This RAS+ output order is not allowed to change and there is no way of specifying such a change in the NIFTI header.

The world in which these RAS+ X, Y, Z axes exist can be one of the recognized spaces, which are: scanner, aligned (to another file's world space), Talairach, MNI 152 (aligned to the MNI 152 atlas).

By implication, the 4th output dimension is likely to be seconds (given the 4th input dimension is likely time), but there's a field `xyzt_units` (see above) that can be used to imply the 4th output dimension is actually frequency, concentration or angular velocity.

NIFTI input / output mapping

NIFTI stores the relationship between the first 3 (spatial) voxel axes and the RAS+ coordinates in an *XYZ affine*. This is a homogeneous coordinate affine, hence 4 by 4 for 3 (spatial) dimensions.

NIFTI also stores "pixel dimensions" in a `pixdim` field. This can give you scaling for individual axes. We ignore the values of `pixdim` for the first 3 axes if we have a full ("sform") affine stored in the header, otherwise they form part of the affine above. `` pixdim``[3:] provide voxel to output scalings for later axes. The units for the 4th dimension can come from ``xyzt_units `` as above.

We take the convention that the output coordinate names are ('x=L->R', 'y=P->A', 'z=I->S','t','u','v','w') unless there is no time axis (see below) in which case we just omit 't'. The first 3 axes are also named after the output space ('scanner-x=L->R', 'mni-x=L-R' etc).

The input axes are 'ijktuvw' unless there is no time axis (see below), in which case they are 'ijkuvw' (remember, NIFTI only allows 7 dimensions, and one is used up by the time length 1 axis).

Time-like axes

A time-like axis is an axis that is any of time, Hz, PPM or radians / second.

We recognize time in a NIPY coordinate map by an input or an output axis named 't' or 'time'. If it's an output axis we work out the corresponding input axis.

A Hz axis can be called 'hz' or 'frequency-hz'.

A PPM axis can be called 'ppm' or 'concentration-ppm'.

A radians / second axis can be called 'rads' or 'radians/s'.

Does this NIFTI image have a time-like axis?

We take there to be no time axis if there are only three NIFTI dimensions, or if:

- the length of the fourth NIFTI dimension is 1 AND
- There are more than four dimensions AND
- The `xyzt_units` field does not indicate time or time-like units.

93.1.2 What we do about all this

For saving a NIPY image to NIFTI, see the docstring for *`nipy2nifti()`*. For loading a NIFTI image to NIPY, see the docstring for *`nifti2nipy()`*.

93.2 Class

93.3 `NiftiError`

class `nipy.io.nifti_ref.`**`NiftiError`**

Bases: `Exception`

`__init__`(**args*, ***kwargs*)

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

93.4 Functions

`nipy.io.nifti_ref.`**`nifti2nipy`**(*ni_img*)

Return NIPY image from NIFTI image *ni_image*

Parameters

ni_img
[nibabel.Nifti1Image] NIFTI image

Returns

img
[`Image`] nipy image

Raises

NiftiError
[if image is < 3D]

Notes

Lacking any other information, we take the input coordinate names for axes 0:7 to be ('i', 'j', 'k', 't', 'u', 'v', 'w').

If the image is 1D or 2D then we have a problem. If there's a defined (sform, qform) affine, this has 3 input dimensions, and we have to guess what the extra input dimensions are. If we don't have a defined affine, we don't know what the output dimensions are. For example, if the image is 2D, and we don't have an affine, are these X and Y or X and Z or Y and Z? In the presence of ambiguity, resist the temptation to guess - raise a NiftiError.

If there is a time-like axis, name the input and corresponding output axis for the type of axis ('t', 'hz', 'ppm', 'rads').

Otherwise remove the 't' axis from both input and output names, and squeeze the length 1 dimension from the input data.

If there's a 't' axis get `toffset` and put into affine at position [3, -1].

If `dim_info` is set coherently, set input axis names to 'slice', 'freq', 'phase' from `dim_info`.

Get the output spatial coordinate names from the 'scanner', 'aligned', 'talairach', 'mni' XYZ spaces (see *`nipy.core.reference.spaces`*).

We construct the N-D affine by taking the XYZ affine and adding scaling diagonal elements from `pixdim`.

If the space units in NIFTI `xyzt_units` are 'microns' or 'meters' we adjust the affine to mm units, but warn because this might be a mistake.

If the time units in NIFTI *xyzt_units* are 'msec' or 'usec', scale the time axis `pixdim` values accordingly.

Ignore the intent-related fields for now, but warn that we are doing so if there appears to be specific information in there.

`nipy.io.nifti_ref.`**`nipy2nifti`**`(`*img*, *data_dtype=None*, *strict=None*, *fix0=True*`)`

Return NIFTI image from nipy image *img*

Parameters

img
[object] An object, usually a NIPY `Image`, having attributes *coordmap* and *shape*

data_dtype
[None or dtype specifier] None means try and use header dtype, otherwise try and use data dtype, otherwise use np.float32. A dtype specifier means set the header output data dtype using `np.dtype(data_dtype)`.

strict
[bool, optional] Whether to use strict checking of input image for creating NIFTI

fix0: bool, optional
Whether to fix potential 0 column / row in affine. This option only used when trying to find time etc axes in the coordmap output names. In order to find matching input names, we need to use the corresponding rows and columns in the affine. Sometimes time, in particular, has 0 scaling, and thus all 0 in the corresponding row / column. In that case it's hard to work out which input corresponds. If *fix0* is True, and there is only one all zero (matrix part of

the) affine row, and only one all zero (matrix part of the) affine column, fix scaling for that combination to zero, assuming this a zero scaling for time.

Returns

ni_img
[`nibabel.Nifti1Image`] NIFTI image

Raises

NiftiError: if space axes not orthogonal to non-space axes
NiftiError: if non-space axes not orthogonal to each other
NiftiError: if *img* output space does not match named spaces in NIFTI
NiftiError: if input image has more than 7 dimensions
NiftiError: if input image has 7 dimensions, but no time dimension, because
we need to add an extra 1 length axis at position 3

NiftiError: if we find a time-like input axis but the matching output axis
is a different time-like.

NiftiError: if we find a time-like output axis but the matching input axis
is a different time-like.

NiftiError: if we find a time output axis and there are non-zero non-spatial
offsets in the affine, but we can't find a corresponding input axis.

Notes

First, we need to create a valid XYZ Affine. We check if this can be done by checking if there are recognizable X, Y, Z output axes and corresponding input (voxel) axes. This requires the input image to be at least 3D. If we find these requirements, we reorder the image axes to have XYZ output axes and 3 spatial input axes first, and get the corresponding XYZ affine.

If the spatial dimensions are not orthogonal to the non-spatial dimensions, raise a NiftiError.

If the non-spatial dimensions are not orthogonal to each other, raise a NiftiError.

We check if the XYZ output fits with the NIFTI named spaces of scanner, aligned, Talairach, MNI. If so, set the NIFTI code and qform, sform accordingly. If the space corresponds to 'unknown' then we must set the NIFTI transform codes to 0, and the affine must match the affine we will get from loading the NIFTI with no qform, sform. If not, we're going to lose information in the affine, and raise an error.

If any of the first three input axes are named ('slice', 'freq', 'phase') set the `dim_info` field accordingly.

Set the `xyzt_units` field to indicate millimeters and seconds, if there is a 't' axis, otherwise millimeters and 0 (unknown).

We look to see if we have a time-like axis in the inputs or the outputs. A time-like axis has labels 't', 'hz', 'ppm', 'rads'. If we have an axis 't' in the inputs *and* the outputs, check they either correspond, or both inputs and output correspond with no other axis, otherwise raise NiftiError. Do the same check for 'hz', then 'ppm', then 'rads'.

If we do have a time-like axis, roll that axis to be the 4th axis. If this axis is actually time, take the `affine[3, -1]` and put into the `toffset` field. If there's no time-like axis, but there are other non-spatial axes, make a length 1 4th array axis to indicate this.

If the resulting NIFTI image has more than 7 dimensions, raise a NiftiError.

Set `pixdim` for axes >= 3 using vector length of corresponding affine columns.

We don't set the intent-related fields for now.

CHAPTER

NINETYFOUR

LABS.DATASETS.CONVERTERS

94.1 Module: `labs.datasets.converters`

Conversion mechanisms for IO and interaction between volumetric datasets and other type of neuroimaging data.

94.2 Functions

`nipy.labs.datasets.converters.`**`as_volume_img`**(*obj*, *copy=True*, *squeeze=True*, *world_space=None*)

Convert the input to a VolumeImg.

Parameters

obj

[filename, pynifti or brifti object, or volume dataset.] Input object, in any form that can be converted to a VolumeImg. This includes Nifti filenames, pynifti or brifti objects, or other volumetric dataset objects.

copy: boolean, optional

If copy is True, the data and affine arrays are copied, elsewhere a view is taken.

squeeze: boolean, optional

If squeeze is True, the data array is squeeze on for dimensions above 3.

world_space: string or None, optional

An optional specification of the world space, to override that given by the image.

Returns

volume_img: VolumeImg object

A VolumeImg object containing the data. The metadata is kept as much as possible in the metadata attribute.

Notes

The world space might not be correctly defined by the input object (in particular, when loading data from disk). In this case, you can correct it manually using the world_space keyword argument.

For pynifti objects, the data is transposed.

`nipy.labs.datasets.converters.`**`save`**(*filename*, *obj*)

Save an nipy image object to a file.

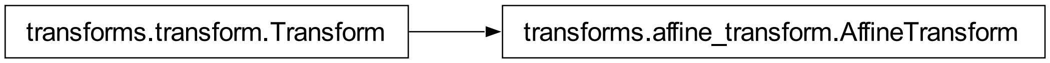
CHAPTER

NINETYFIVE

LABS.DATASETS.TRANSFORMS.AFFINE_TRANSFORM

95.1 Module: `labs.datasets.transforms.affine_transform`

Inheritance diagram for `nipy.labs.datasets.transforms.affine_transform`:


transforms.transform.Transform
transforms.affine_transform.AffineTransform

The AffineTransform class

95.2 `AffineTransform`

class `nipy.labs.datasets.transforms.affine_transform.`**`AffineTransform`**(*input_space*, *output_space*, *affine*)

Bases: *`Transform`*

A transformation from an input 3D space to an output 3D space defined by an affine matrix.

It is defined by the affine matrix , and the name of the input and output spaces.

`__init__`(*input_space*, *output_space*, *affine*)

Create a new affine transform object.

Parameters

input_space: string
Name of the input space

output_space: string
Name of the output space

affine: 4x4 ndarray
Affine matrix giving the coordinate mapping between the input and output space.

`affine = None`

`composed_with`(*transform*)

Returns a new transform obtained by composing this transform with the one provided.

Parameters

transform: nipy.core.transforms.transform object
The transform to compose with.

`get_inverse`()

Return the inverse transform.

`input_space = ''`

`inverse_mapping`(*x*, *y*, *z*)

Transform the given coordinate from output space to input space.

Parameters

x: number or ndarray
The x coordinates

y: number or ndarray
The y coordinates

z: number or ndarray
The z coordinates

`mapping`(*x*, *y*, *z*)

Transform the given coordinate from input space to output space.

Parameters

x: number or ndarray
The x coordinates

y: number or ndarray
The y coordinates

z: number or ndarray
The z coordinates

`output_space = ''`

CHAPTER

NINETYSIX

LABS.DATASETS.TRANSFORMS.AFFINE_UTILS

96.1 Module: `labs.datasets.transforms.affine_utils`

Functions working with affine transformation matrices.

96.2 Functions

`nipy.labs.datasets.transforms.affine_utils.`**`apply_affine`**(*x*, *y*, *z*, *affine*)

Apply the affine matrix to the given coordinate.

Parameters

x: number or ndarray
The x coordinates

y: number or ndarray
The y coordinates

z: number or ndarray
The z coordinates

affine: 4x4 ndarray
The affine matrix of the transformation

`nipy.labs.datasets.transforms.affine_utils.`**`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`

`nipy.labs.datasets.transforms.affine_utils.`**`get_bounds`**`(`*shape*, *affine*`)`

Return the world-space bounds occupied by an array given an affine.

`nipy.labs.datasets.transforms.affine_utils.`**`to_matrix_vector`**`(`*transform*`)`

Split a transform into it's matrix and vector components.

The transformation 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`

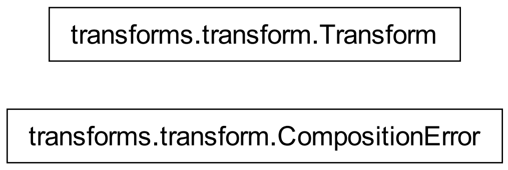
CHAPTER

NINETYSEVEN

LABS.DATASETS.TRANSFORMS.TRANSFORM

97.1 Module: `labs.datasets.transforms.transform`

Inheritance diagram for `nipy.labs.datasets.transforms.transform`:


transforms.transform.Transform
transforms.transform.CompositionError

The base Transform class.

This class defines the Transform interface and can be subclassed to define more clever composition logic.

97.2 Classes

97.2.1 `CompositionError`

class `nipy.labs.datasets.transforms.transform.`**`CompositionError`**

Bases: `Exception`

The Exception raised when composing transforms with non matching respective input and output word spaces.

`__init__`(**args*, ***kwargs*)

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

97.2.2 `Transform`

class `nipy.labs.datasets.transforms.transform.`**`Transform`**(*input_space*, *output_space*, *mapping=None*, *inverse_mapping=None*)

Bases: `object`

A transform is a representation of a transformation from one 3D space to another. It is composed of a coordinate mapping, or its inverse, as well as the name of the input and output spaces.

The Transform class is the base class for transformations and defines the transform object API.

`__init__`(*input_space*, *output_space*, *mapping=None*, *inverse_mapping=None*)

Create a new transform object.

Parameters

mapping: callable f(x, y, z)
Callable mapping coordinates from the input space to the output space. It should take 3 numbers or arrays, and return 3 numbers or arrays of the same shape.

inverse_mapping: callable f(x, y, z)
Callable mapping coordinates from the output space to the input space. It should take 3 numbers or arrays, and return 3 numbers or arrays of the same shape.

input_space: string
Name of the input space

output_space: string
Name of the output space

Notes

You need to supply either the mapping or the inverse mapping.

`composed_with`(*transform*)

Returns a new transform obtained by composing this transform with the one provided.

Parameters

transform: nipy.core.transforms.transform object
The transform to compose with.

`get_inverse`()

Return the inverse transform.

`input_space = ''`

`inverse_mapping = None`

`mapping = None`

`output_space = ''`

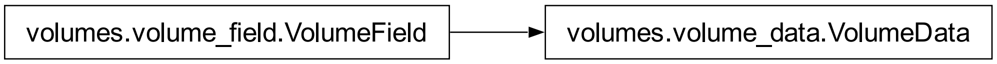
CHAPTER

NINETYEIGHT

LABS.DATASETS.VOLUMES.VOLUME_DATA

98.1 Module: `labs.datasets.volumes.volume_data`

Inheritance diagram for `nipy.labs.datasets.volumes.volume_data`:


volumes.volume_field.VolumeField
volumes.volume_data.VolumeData

The volume data class

This class represents indexable data embedded in a 3D space

98.2 `VolumeData`

class `nipy.labs.datasets.volumes.volume_data.`**`VolumeData`**

Bases: *`VolumeField`*

A class representing data embedded in a 3D space

This object has data stored in an array like, that knows how it is mapped to a 3D "real-world space", and how it can change real-world coordinate system.

Notes

The data is stored in an undefined way: prescalings might need to be applied to it before using it, or the data might be loaded on demand. The best practice to access the data is not to access the _data attribute, but to use the *get_fdata* method.

Attributes

world_space: string
World space the data is embedded in. For instance *mni152*.

metadata: dictionary
Optional, user-defined, dictionary used to carry around extra information about the data as

it goes through transformations. The class consistency of this information is not maintained as the data is modified.

_data:
Private pointer to the data.

`__init__`(**args*, ***kwargs*)

`as_volume_img`(*affine=None*, *shape=None*, *interpolation=None*, *copy=True*)

Resample the image to be an image with the data points lying on a regular grid with an affine mapping to the word space (a nipy VolumeImg).

Parameters

affine: 4x4 or 3x3 ndarray, optional
Affine of the new voxel grid or transform object pointing to the new voxel coordinate grid. If a 3x3 ndarray is given, it is considered to be the rotation part of the affine, and the best possible bounding box is calculated, in this case, the shape argument is not used. If None is given, a default affine is provided by the image.

shape: (n_x, n_y, n_z), tuple of integers, optional
The shape of the grid used for sampling, if None is given, a default affine is provided by the image.

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

resampled_image
[nipy VolumeImg] New nipy VolumeImg with the data sampled on the grid defined by the affine and shape.

Notes

The coordinate system of the image is not changed: the returned image points to the same world space.

`composed_with_transform`(*w2w_transform*)

Return a new image embedding the same data in a different word space using the given world to world transform.

Parameters

w2w_transform
[transform object] The transform object giving the mapping between the current world space of the image, and the new word space.

Returns

remapped_image
[nipy image] An image containing the same data, expressed in the new world space.

`get_fdata`()

Return data as a numpy array.

`get_transform`()

Returns the transform object associated with the volumetric structure which is a general description of the mapping from the values to the world space.

Returns

transform
[nipy.datasets.Transform object]

`interpolation = 'continuous'`

`like_from_data`(*data*)

Returns an volumetric data structure with the same relationship between data and world space, and same metadata, but different data.

Parameters

data: ndarray

`metadata = {}`

`resampled_to_img`(*target_image*, *interpolation=None*)

Resample the data to be on the same voxel grid than the target volume structure.

Parameters

target_image
[nipy image] Nipy image onto the voxel grid of which the data will be resampled. This can be any kind of img understood by Nipy (datasets, pynifti objects, nibabel object) or a string giving the path to a nifti of analyse image.

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

resampled_image
[nipy_image] New nipy image with the data resampled.

Notes

Both the target image and the original image should be embedded in the same world space.

`values_in_world`(*x*, *y*, *z*, *interpolation=None*)

Return the values of the data at the world-space positions given by x, y, z

Parameters

x
[number or ndarray] x positions in world space, in other words millimeters

y
[number or ndarray] y positions in world space, in other words millimeters. The shape of y should match the shape of x

z
[number or ndarray] z positions in world space, in other words millimeters. The shape of z should match the shape of x

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

values
[number or ndarray] Data values interpolated at the given world position. This is a number or an ndarray, depending on the shape of the input coordinate.

`world_space = ''`

CHAPTER

NINETYNINE

LABS.DATASETS.VOLUMES.VOLUME_FIELD

99.1 Module: `labs.datasets.volumes.volume_field`

Inheritance diagram for `nipy.labs.datasets.volumes.volume_field`:

The base volumetric field interface

This defines the nipy volumetric structure interface.

99.2 `VolumeField`

class `nipy.labs.datasets.volumes.volume_field.`**`VolumeField`**

Bases: `object`

The base volumetric structure.

This object represents numerical values embedded in a 3-dimensional world space (called a field in physics and engineering)

This is an abstract base class: it defines the interface, but not the logics.

Attributes

world_space: string
World space the data is embedded in. For instance *mni152*.

metadata: dictionary
Optional, user-defined, dictionary used to carry around extra information about the data as it goes through transformations. The consistency of this information is not maintained as the data is modified.

`__init__`(**args*, ***kwargs*)

`as_volume_img`(*affine=None*, *shape=None*, *interpolation=None*, *copy=True*)

Resample the image to be an image with the data points lying on a regular grid with an affine mapping to the word space (a nipy VolumeImg).

Parameters

affine: 4x4 or 3x3 ndarray, optional
Affine of the new voxel grid or transform object pointing to the new voxel coordinate grid. If a 3x3 ndarray is given, it is considered to be the rotation part of the affine, and the best possible bounding box is calculated, in this case, the shape argument is not used. If None is given, a default affine is provided by the image.

shape: (n_x, n_y, n_z), tuple of integers, optional
The shape of the grid used for sampling, if None is given, a default affine is provided by the image.

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

resampled_image
[nipy VolumeImg] New nipy VolumeImg with the data sampled on the grid defined by the affine and shape.

Notes

The coordinate system of the image is not changed: the returned image points to the same world space.

`composed_with_transform`(*w2w_transform*)

Return a new image embedding the same data in a different word space using the given world to world transform.

Parameters

w2w_transform
[transform object] The transform object giving the mapping between the current world space of the image, and the new word space.

Returns

remapped_image
[nipy image] An image containing the same data, expressed in the new world space.

`get_transform`()

Returns the transform object associated with the volumetric structure which is a general description of the mapping from the values to the world space.

Returns

transform
[nipy.datasets.Transform object]

`metadata = {}`

`resampled_to_img`(*target_image*, *interpolation=None*)

Resample the volume to be sampled similarly than the target volumetric structure.

Parameters

target_image
[nipy volume] Nipy volume structure onto the grid of which the data will be resampled.

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the volume's interpolation logic is used.

Returns

resampled_image
[nipy_image] New nipy image with the data resampled.

Notes

Both the target image and the original image should be embedded in the same world space.

`values_in_world`(*x*, *y*, *z*, *interpolation=None*)

Return the values of the data at the world-space positions given by x, y, z

Parameters

x
[number or ndarray] x positions in world space, in other words millimeters

y
[number or ndarray] y positions in world space, in other words millimeters. The shape of y should match the shape of x

z
[number or ndarray] z positions in world space, in other words millimeters. The shape of z should match the shape of x

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

values
[number or ndarray] Data values interpolated at the given world position. This is a number or an ndarray, depending on the shape of the input coordinate.

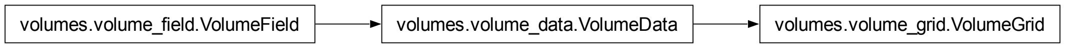
`world_space = ''`

CHAPTER

LABS.DATASETS.VOLUMES.VOLUME_GRID

100.1 Module: `labs.datasets.volumes.volume_grid`

Inheritance diagram for `nipy.labs.datasets.volumes.volume_grid`:


volumes.volume_field.VolumeField
volumes.volume_data.VolumeData
volumes.volume_grid.VolumeGrid

The volume grid class.

This class represents data lying on a (non rigid, non regular) grid embedded in a 3D world represented as a 3+D array.

100.2 `VolumeGrid`

class `nipy.labs.datasets.volumes.volume_grid.`**`VolumeGrid`**(*data*, *transform*, *metadata=None*, *interpolation='continuous'*)

Bases: *`VolumeData`*

A class representing data stored in a 3+D array embedded in a 3D world.

This object has data stored in an array-like multidimensional indexable objects, with the 3 first dimensions corresponding to spatial axis and defining a 3D grid that may be non-regular or non-rigid.

The object knows how the data is mapped to a 3D "real-world space", and how it can change real-world coordinate system. The transform mapping it to world is arbitrary, and thus the grid can be warped: in the world space, the grid may not be regular or orthogonal.

Notes

The data is stored in an undefined way: prescalings might need to be applied to it before using it, or the data might be loaded on demand. The best practice to access the data is not to access the _data attribute, but to use the *get_fdata* method.

If the transform associated with the image has no inverse mapping, data corresponding to a given world space position cannot be calculated. If it has no forward mapping, it is impossible to resample another dataset on the same support.

Attributes

world_space: string
World space the data is embedded in. For instance *mni152*.

metadata: dictionary
Optional, user-defined, dictionary used to carry around extra information about the data as it goes through transformations. The consistency of this information is not maintained as the data is modified.

_data:
Private pointer to the data.

`__init__`(*data*, *transform*, *metadata=None*, *interpolation='continuous'*)

The base image containing data.

Parameters

data: ndarray
n dimensional array giving the embedded data, with the 3 first dimensions being spatial.

transform: nipy transform object
The transformation from voxel to world.

metadata
[dictionary, optional] Dictionary of user-specified information to store with the image.

interpolation
['continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces.

`as_volume_img`(*affine=None*, *shape=None*, *interpolation=None*, *copy=True*)

Resample the image to be an image with the data points lying on a regular grid with an affine mapping to the word space (a nipy VolumeImg).

Parameters

affine: 4x4 or 3x3 ndarray, optional
Affine of the new voxel grid or transform object pointing to the new voxel coordinate grid. If a 3x3 ndarray is given, it is considered to be the rotation part of the affine, and the best possible bounding box is calculated, in this case, the shape argument is not used. If None is given, a default affine is provided by the image.

shape: (n_x, n_y, n_z), tuple of integers, optional
The shape of the grid used for sampling, if None is given, a default affine is provided by the image.

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

resampled_image
[nipy VolumeImg] New nipy VolumeImg with the data sampled on the grid defined by the affine and shape.

Notes

The coordinate system of the image is not changed: the returned image points to the same world space.

`composed_with_transform`(*w2w_transform*)

Return a new image embedding the same data in a different word space using the given world to world transform.

Parameters

w2w_transform
[transform object] The transform object giving the mapping between the current world space of the image, and the new word space.

Returns

remapped_image
[nipy image] An image containing the same data, expressed in the new world space.

`get_fdata`()

Return data as a numpy array.

`get_transform`()

Returns the transform object associated with the volumetric structure which is a general description of the mapping from the values to the world space.

Returns

transform
[nipy.datasets.Transform object]

`get_world_coords`()

Return the data points coordinates in the world space.

Returns

x: ndarray
x coordinates of the data points in world space

y: ndarray
y coordinates of the data points in world space

z: ndarray
z coordinates of the data points in world space

`interpolation = 'continuous'`

`like_from_data`(*data*)

Returns an volumetric data structure with the same relationship between data and world space, and same metadata, but different data.

Parameters

data: ndarray

`metadata = {}`

`resampled_to_img`(*target_image*, *interpolation=None*)

Resample the data to be on the same voxel grid than the target volume structure.

Parameters

target_image
[nipy image] Nipy image onto the voxel grid of which the data will be resampled. This can be any kind of img understood by Nipy (datasets, pynifti objects, nibabel object) or a string giving the path to a nifti of analyse image.

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

resampled_image
[nipy_image] New nipy image with the data resampled.

Notes

Both the target image and the original image should be embedded in the same world space.

`values_in_world`(*x*, *y*, *z*, *interpolation=None*)

Return the values of the data at the world-space positions given by x, y, z

Parameters

x
[number or ndarray] x positions in world space, in other words millimeters

y
[number or ndarray] y positions in world space, in other words millimeters. The shape of y should match the shape of x

z
[number or ndarray] z positions in world space, in other words millimeters. The shape of z should match the shape of x

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

values
[number or ndarray] Data values interpolated at the given world position. This is a number or an ndarray, depending on the shape of the input coordinate.

`world_space = ''`

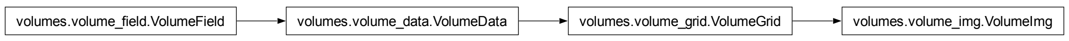
CHAPTER

ONE

LABS.DATASETS.VOLUMES.VOLUME_IMG

101.1 Module: `labs.datasets.volumes.volume_img`

Inheritance diagram for `nipy.labs.datasets.volumes.volume_img`:


volumes.volume_field.VolumeField
volumes.volume_data.VolumeData
volumes.volume_grid.VolumeGrid
volumes.volume_img.VolumeImg

An image that stores the data as an (x, y, z, . . .) array, with an affine mapping to the world space

101.2 `VolumeImg`

class `nipy.labs.datasets.volumes.volume_img.`**`VolumeImg`**(*data*, *affine*, *world_space*, *metadata=None*, *interpolation='continuous'*)

Bases: *`VolumeGrid`*

A regularly-spaced image for embedding data in an x, y, z 3D world, for neuroimaging.

This object is an ndarray representing a volume, with the first 3 dimensions being spatial, and mapped to a named world space using an affine (4x4 matrix).

Notes

The data is stored in an undefined way: prescalings might need to be applied to it before using it, or the data might be loaded on demand. The best practice to access the data is not to access the _data attribute, but to use the *get_fdata* method.

Attributes

affine
[4x4 ndarray] Affine mapping from indices to world coordinates.

world_space
[string] Name of the world space the data is embedded in. For instance *mni152*.

metadata
[dictionary] Optional, user-defined, dictionary used to carry around extra information about

the data as it goes through transformations. The consistency of this information may not be maintained as the data is modified.

interpolation
['continuous' or 'nearest'] String giving the interpolation logic used when calculating values in different world spaces

_data
Private pointer to the data.

`__init__`(*data*, *affine*, *world_space*, *metadata=None*, *interpolation='continuous'*)

Creates a new neuroimaging image with an affine mapping.

Parameters

data
[ndarray] ndarray representing the data.

affine
[4x4 ndarray] affine transformation to the reference world space

world_space
[string] name of the reference world space.

metadata
[dictionary] dictionary of user-specified information to store with the image.

`affine = array([[1., 0., 0., 0.], [0., 1., 0., 0.], [0., 0., 1., 0.], [0., 0., 0., 1.]])`

`as_volume_img`(*affine=None*, *shape=None*, *interpolation=None*, *copy=True*)

Resample the image to be an image with the data points lying on a regular grid with an affine mapping to the word space (a nipy VolumeImg).

Parameters

affine: 4x4 or 3x3 ndarray, optional
Affine of the new voxel grid or transform object pointing to the new voxel coordinate grid. If a 3x3 ndarray is given, it is considered to be the rotation part of the affine, and the best possible bounding box is calculated, in this case, the shape argument is not used. If None is given, a default affine is provided by the image.

shape: (n_x, n_y, n_z), tuple of integers, optional
The shape of the grid used for sampling, if None is given, a default affine is provided by the image.

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

resampled_image
[nipy VolumeImg] New nipy VolumeImg with the data sampled on the grid defined by the affine and shape.

Notes

The coordinate system of the image is not changed: the returned image points to the same world space.

`composed_with_transform`(*w2w_transform*)

Return a new image embedding the same data in a different word space using the given world to world transform.

Parameters

w2w_transform
[transform object] The transform object giving the mapping between the current world space of the image, and the new word space.

Returns

remapped_image
[nipy image] An image containing the same data, expressed in the new world space.

`get_affine`()

`get_fdata`()

Return data as a numpy array.

`get_transform`()

Returns the transform object associated with the volumetric structure which is a general description of the mapping from the values to the world space.

Returns

transform
[nipy.datasets.Transform object]

`get_world_coords`()

Return the data points coordinates in the world space.

Returns

x: ndarray
x coordinates of the data points in world space

y: ndarray
y coordinates of the data points in world space

z: ndarray
z coordinates of the data points in world space

`interpolation = 'continuous'`

`like_from_data`(*data*)

Returns an volumetric data structure with the same relationship between data and world space, and same metadata, but different data.

Parameters

data: ndarray

`metadata = {}`

`resampled_to_img`(*target_image*, *interpolation=None*)

Resample the data to be on the same voxel grid than the target volume structure.

Parameters

target_image
[nipy image] Nipy image onto the voxel grid of which the data will be resampled. This can be any kind of img understood by Nipy (datasets, pynifti objects, nibabel object) or a string giving the path to a nifti of analyse image.

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

resampled_image
[nipy_image] New nipy image with the data resampled.

Notes

Both the target image and the original image should be embedded in the same world space.

`values_in_world`(*x*, *y*, *z*, *interpolation=None*)

Return the values of the data at the world-space positions given by x, y, z

Parameters

x
[number or ndarray] x positions in world space, in other words millimeters

y
[number or ndarray] y positions in world space, in other words millimeters. The shape of y should match the shape of x

z
[number or ndarray] z positions in world space, in other words millimeters. The shape of z should match the shape of x

interpolation
[None, 'continuous' or 'nearest', optional] Interpolation type used when calculating values in different word spaces. If None, the image's interpolation logic is used.

Returns

values
[number or ndarray] Data values interpolated at the given world position. This is a number or an ndarray, depending on the shape of the input coordinate.

`world_space = ''`

`xyz_ordered`(*resample=False*, *copy=True*)

Returns an image with the affine diagonal and positive in the world space it is embedded in.

Parameters

resample: boolean, optional
If resample is False, no resampling is performed, the axis are only permuted. If it is impossible to get xyz ordering by permuting the axis, a 'CompositionError' is raised.

copy: boolean, optional
If copy is True, a deep copy of the image (including the data) is made.

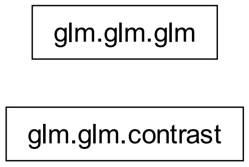
CHAPTER

TWO

LABS.GLM.GLM

102.1 Module: `labs.glm.glm`

Inheritance diagram for `nipy.labs.glm.glm`:



102.2 Classes

102.2.1 `contrast`

class `nipy.labs.glm.glm.`**`contrast`**(*dim*, *type='t'*, *tiny=1e-50*, *dofmax=10000000000.0*)

Bases: `object`

`__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'

102.2.2 glm

class nipy.labs.glm.glm.**glm**(*Y=None*, *X=None*, *formula=None*, *axis=0*, *model='spherical'*, *method=None*, *niter=2*)

Bases: `object`

__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*)

Specify and estimate a contrast

c must be a numpy.ndarray (or anything that numpy.asarray can cast to a ndarray). For a F contrast, c must be q x p where q is the number of contrast vectors and p is the total number of regressors.

fit(*Y*, *X*, *formula=None*, *axis=0*, *model='spherical'*, *method=None*, *niter=2*)

save(*file*)

Save fit into a .npz file

102.3 Functions

nipy.labs.glm.glm.**load**(*file*)

Load a fitted glm

nipy.labs.glm.glm.**ols**(*Y*, *X*, *axis=0*)

Essentially, compute pinv(X)*Y

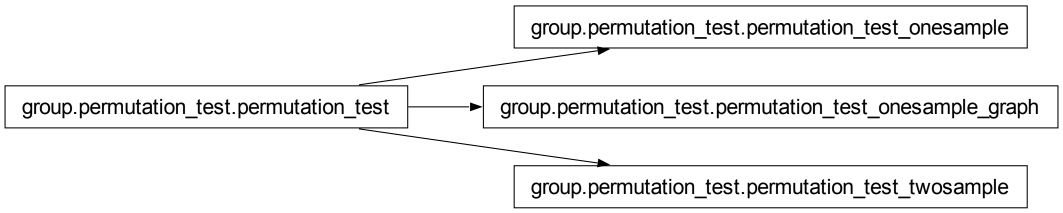
CHAPTER

THREE

LABS.GROUP.PERMUTATION_TEST

103.1 Module: `labs.group.permutation_test`

Inheritance diagram for `nipy.labs.group.permutation_test`:


group.permutation_test.permutation_test
group.permutation_test.permutation_test_onesample
group.permutation_test.permutation_test_onesample_graph
group.permutation_test.permutation_test_twosample

One and two sample permutation tests.

103.2 Classes

103.2.1 `permutation_test`

class `nipy.labs.group.permutation_test.`**`permutation_test`**

Bases: `object`

This generic permutation test class contains the calibration method which is common to the derived classes permutation_test_onesample and permutation_test_twosample (as well as other common methods)

`__init__`(**args*, ***kwargs*)

`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.

103.2.2 permutation_test_onesample

`class nipy.labs.group.permutation_test.`**`permutation_test_onesample`**(*data*, *XYZ*, *axis=0*, *vardata=None*, *stat_id='student'*, *base=0.0*, *niter=5*, *ndraws=100000*)

Bases: *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=100000*)

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

`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.

103.2.3 `permutation_test_onesample_graph`

`class nipy.labs.group.permutation_test.`**`permutation_test_onesample_graph`**(*data*, *G*, *axis=0*, *vardata=None*, *stat_id='student'*, *base=0.0*, *niter=5*, *ndraws=100000*)

Bases: `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=100000*)

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

`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.

103.2.4 permutation_test_twosample

class nipy.labs.group.permutation_test.**permutation_test_twosample**(*data1*, *data2*, *XYZ*, *axis=0*, *vardata1=None*, *vardata2=None*, *stat_id='student'*, *niter=5*, *ndraws=100000*)

Bases: *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=100000*)

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

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.

103.3 Functions

`nipy.labs.group.permutation_test.`**`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

`nipy.labs.group.permutation_test.`**`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

`nipy.labs.group.permutation_test.`**`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

Comment by alexis-roche, September 15th 2012: this function was originally developed by Merlin Keller in an attempt to generalize classical cluster-level analysis by subdividing clusters in blobs with limited diameter (at least, this is my understanding). This piece of code seems to have remained very experimental and its usefulness in real-world neuroimaging image studies is still to be demonstrated.

`nipy.labs.group.permutation_test.`**`extract_clusters_from_graph`**`(`*T*, *G*, *th*`)`

This returns a label vector of same size as T, defining connected components for subgraph of weighted graph G containing vertices s.t. T >= th

`nipy.labs.group.permutation_test.`**`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

`nipy.labs.group.permutation_test.`**`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>

`nipy.labs.group.permutation_test.`**`onesample_stat`**`(`*Y*, *V*, *stat_id*, *base=0.0*, *axis=0*, *Magics=None*, *niter=5*`)`

Wrapper for os_stat and os_stat_mfx

`nipy.labs.group.permutation_test.`**`peak_XYZ`**`(`*XYZ*, *Tvalues*, *labels*, *label_values*`)`

Returns (3, n_labels) array of maximum T values coordinates for each label value

`nipy.labs.group.permutation_test.`**`sorted_values`**`(`*a*`)`

Extract list of distinct sortedvalues from an array

`nipy.labs.group.permutation_test.`**`twosample_stat`**`(`*Y1*, *V1*, *Y2*, *V2*, *stat_id*, *axis=0*, *Magics=None*, *niter=5*`)`

Wrapper for ts_stat and ts_stat_mfx

CHAPTER

FOUR

LABS.MASK

104.1 Module: `labs.mask`

Utilities for extracting masks from EPI images and applying them to time series.

104.2 Functions

`nipy.labs.mask.`**`compute_mask`**`(`*mean_volume*, *reference_volume=None*, *m=0.2*, *M=0.9*, *cc=True*, *opening=2*, *exclude_zeros=False*`)`

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.

opening: int, optional
if opening is larger than 0, an morphological opening is performed, to keep only large structures. This step is useful to remove parts of the skull that might have been included.

exclude_zeros: boolean, optional
Consider zeros as missing values for the computation of the threshold. This option is useful if the images have been resliced with a large padding of zeros.

Returns

mask
[3D boolean ndarray] The brain mask

`nipy.labs.mask.compute_mask_files`(*input_filename*, *output_filename=None*, *return_mean=False*, *m=0.2*, *M=0.9*, *cc=1*, *exclude_zeros=False*, *opening=2*)

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).

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.

exclude_zeros: boolean, optional
Consider zeros as missing values for the computation of the threshold. This option is useful if the images have been resliced with a large padding of zeros.

opening: int, optional
Size of the morphological opening performed as post-processing

Returns

mask
[3D boolean array] 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.

`nipy.labs.mask.compute_mask_sessions`(*session_images*, *m=0.2*, *M=0.9*, *cc=1*, *threshold=0.5*, *exclude_zeros=False*, *return_mean=False*, *opening=2*)

Compute a common mask for several sessions of fMRI data.

Uses the mask-finding algorithms 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_images
[list of (list of strings) or nipy image objects] A list of images/list of nifti filenames. Each inner list/image represents a session.

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.

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.

exclude_zeros: boolean, optional
Consider zeros as missing values for the computation of the threshold. This option is useful if the images have been resliced with a large padding of zeros.

return_mean: boolean, optional
if return_mean is True, the mean image across subjects is returned.

opening: int, optional,
size of the morphological opening

Returns

mask
[3D boolean ndarray] The brain mask

mean
[3D float array] The mean image

`nipy.labs.mask.`**`intersect_masks`**`(`*input_masks*, *output_filename=None*, *threshold=0.5*, *cc=True*`)`

Given a list of input mask images, generate the output image which is the the threshold-level intersection of the inputs

Parameters

input_masks: list of strings or ndarrays
paths of the input images nsubj set as len(input_mask_files), or individual masks.

output_filename, string:
Path of the output image, if None no file is saved.

threshold: float within [0, 1[, optional
gives the level of the intersection. threshold=1 corresponds to keeping the intersection of all masks, whereas threshold=0 is the union of all masks.

cc: bool, optional
If true, extract the main connected component

Returns

grp_mask, boolean array of shape the image shape

`nipy.labs.mask.`**`largest_cc`**`(`*mask*`)`

Return the largest connected component of a 3D mask array.

Parameters

mask: 3D boolean array
3D array indicating a mask.

Returns

mask: 3D boolean array
3D array indicating a mask, with only one connected component.

`nipy.labs.mask.`**`series_from_mask`**(*filenames*, *mask*, *dtype=<class 'numpy.float32'>*, *smooth=False*, *ensure_finite=True*)

Read the time series from the given sessions filenames, using the mask.

Parameters

filenames: list of 3D nifti file names, or 4D nifti filename.
Files are grouped by session.

mask: 3d ndarray
3D mask array: true where a voxel should be used.

smooth: False or float, optional
If smooth is not False, it gives the size, in voxel of the spatial smoothing to apply to the signal.

ensure_finite: boolean, optional
If ensure_finite is True, the non-finite values (NaNs and infs) found in the images will be replaced by zeros

Returns

session_series: ndarray
3D array of time course: (session, voxel, time)

header: header object
The header of the first file.

Notes

When using smoothing, ensure_finite should be True: as elsewhere non finite values will spread across the image.

`nipy.labs.mask.`**`threshold_connect_components`**(*map*, *threshold*, *copy=True*)

Given a map with some coefficients set to zero, segment the
connect components with number of voxels smaller than the threshold and set them to 0.

Parameters

map: ndarray,
The spatial map to segment

threshold: scalar,
The minimum number of voxels to keep a cluster.

copy: bool, optional
If copy is false, the input array is modified inplace

Returns

map: ndarray,
the map with connected components removed

CHAPTER

FIVE

LABS.SPATIAL_MODELS.BAYESIAN_STRUCTURAL_ANALYSIS

105.1 Module: `labs.spatial_models.bayesian_structural_analysis`

The main routine of this package that aims at performing the extraction of ROIs from multisubject dataset using the localization and activation strength of extracted regions.

This has been published in: - Thirion et al. High level group analysis of FMRI data based on Dirichlet process mixture models, IPMI 2007 - Thirion et al. Accurate Definition of Brain Regions Position Through the Functional Landmark Approach, MICCAI 2010

Author : Bertrand Thirion, 2006-2013

`nipy.labs.spatial_models.bayesian_structural_analysis.`**`compute_landmarks`**(*domain*, *stats*, *sigma*, *prevalence_pval=0.5*, *prevalence_threshold=0*, *threshold=3.0*, *smin=5*, *method='prior'*, *algorithm='density'*, *n_iter=1000*, *burnin=100*)

Compute the Bayesian Structural Activation patterns

Parameters

domain: StructuredDomain instance,
Description of the spatial context of the data

stats: array of shape (nbnodes, subjects):
the multi-subject statistical maps

sigma: float > 0:
expected cluster std in the common space in units of coord

prevalence_pval: float in the [0,1] interval, optional
posterior significance threshold

prevalence_threshold: float, optional,
reference threshold for the prevalence value

threshold: float, optional,
first level threshold

smin: int, optional,
minimal size of the regions to validate them

method: {'gauss_mixture', 'emp_null', 'gam_gauss', 'prior'}, optional,
'gauss_mixture' A Gaussian Mixture Model is used 'emp_null' a null mode is fitted to test 'gam_gauss' a Gamma-Gaussian mixture is used 'prior' a hard-coded function is used

algorithm: string, one of ['density', 'co-occurrence'], optional
method used to compute the landmarks

niter: int, optional,
number of iterations of the DPMM

burnin: int, optional,
number of iterations of the DPMM

Returns

landmarks: Instance of sbf.LandmarkRegions or None,
Describes the ROIs found in inter-subject inference None if nothing can be defined

hrois: list of nipy.labs.spatial_models.hroi.Nroi instances
representing individual ROIs

CHAPTER

SIX

LABS.SPATIAL_MODELS.BSA_IO

106.1 Module: `labs.spatial_models.bsa_io`

This module is the interface to the bayesian_structural_analysis (bsa) module It handles the images provided as input and produces result images.

`nipy.labs.spatial_models.bsa_io.`**`make_bsa_image`**(*mask_images*, *stat_images*, *threshold=3.0*, *smin=0*, *sigma=5.0*, *prevalence_threshold=0*, *prevalence_pval=0.5*, *write_dir=None*, *algorithm='density'*, *contrast_id='default'*)

Main function for performing bsa on a set of images. It creates the some output images in the given directory

Parameters

mask_images: list of str,
image paths that yield mask images, one for each subject

stat_images: list of str,
image paths of the activation images, one for each subject

threshold: float, optional,
threshold used to ignore all the image data that is below

smin: float, optional,
minimal size (in voxels) of the extracted blobs smaller blobs are merged into larger ones

sigma: float, optional,
variance of the spatial model, i.e. cross-subject uncertainty

prevalence_threshold: float, optional
threshold on the representativity measure

prevalence_pval: float, optional,

p-value of the representativity test:

test = p(representativity>prevalence_threshold) > prevalence_pval

write_dir: string, optional,
if not None, output directory

method: {'density', 'co-occurrence'}, optional,
Inference method used in the landmark definition

contrast_id: string, optional,
identifier of the contrast

Returns

landmarks: nipy.labs.spatial_models.structural_bfls.landmark_regions
instance that describes the structures found at the group level None is returned if nothing has been found significant at the group level

hrois
[list of nipy.labs.spatial_models.hroi.Nroi instances,] (one per subject), describe the individual counterpart of landmarks

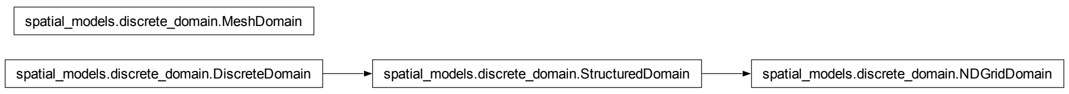
CHAPTER

SEVEN

LABS.SPATIAL_MODELS.DISCRETE_DOMAIN

107.1 Module: `labs.spatial_models.discrete_domain`

Inheritance diagram for `nipy.labs.spatial_models.discrete_domain`:


spatial_models.discrete_domain.MeshDomain
spatial_models.discrete_domain.DiscreteDomain
spatial_models.discrete_domain.StructuredDomain
spatial_models.discrete_domain.NDGridDomain

This module defines the StructuredDomain class, that represents a generic neuroimaging kind of domain This is meant to provide a unified API to deal with n-d imaged and meshes.

Author: Bertrand Thirion, 2010

107.2 Classes

107.2.1 `DiscreteDomain`

class `nipy.labs.spatial_models.discrete_domain.`**`DiscreteDomain`**(*dim*, *coord*, *local_volume*, *id=''*, *referential=''*)

Bases: `object`

Descriptor of a certain domain that consists of discrete elements that are characterized by a coordinate system and a topology: the coordinate system is specified through a coordinate array the topology encodes the neighboring system

`__init__`(*dim*, *coord*, *local_volume*, *id=''*, *referential=''*)

Initialize discrete domain instance

Parameters

dim: int
the (physical) dimension of the domain.

coord: array of shape(size, em_dim)
explicit coordinates of the domain sites.

local_volume: array of shape(size)
yields the volume associated with each site.

id: string, optional
domain identifier.

referential: string, optional
identifier of the referential of the coordinates system.

Notes

Caveat: em_dim may be greater than dim e.g. (meshes coordinate in 3D)

`connected_components()`

returns a labelling of the domain into connected components

`copy()`

Returns a copy of self

`get_coord()`

Returns self.coord

`get_feature(`*fid*`)`

Return self.features[fid]

`get_volume()`

Returns self.local_volume

`integrate(`*fid*`)`

Integrate certain feature over the domain and returns the result

Parameters

fid
[string, feature identifier,] by default, the 1 function is integrataed, yielding domain volume

Returns

lsum = array of shape (self.feature[fid].shape[1]),
the result

`mask(`*bmask*, *id=''*`)`

Returns an DiscreteDomain instance that has been further masked

`representative_feature(`*fid*, *method*`)`

Compute a statistical representative of the within-Foain feature

Parameters

fid: string, feature id
method: string, method used to compute a representative
to be chosen among 'mean', 'max', 'median', 'min'

`set_feature(`*fid*, *data*, *override=True*`)`

Append a feature 'fid'

Parameters

fid: string,
feature identifier

data: array of shape(self.size, p) or self.size
the feature data

107.2.2 MeshDomain

class nipy.labs.spatial_models.discrete_domain.**MeshDomain**(*coord*, *triangles*)

Bases: `object`

temporary class to handle meshes

__init__(*coord*, *triangles*)

Initialize mesh domain instance

Parameters

coord: array of shape (n_vertices, 3),
the node coordinates

triangles: array of shape(n_triables, 3),
indices of the nodes per triangle

area()

Return array of areas for each node

Returns

area: array of shape self.V,
area of each node

topology()

Returns a sparse matrix that represents the connectivity in self

107.2.3 NDGridDomain

class nipy.labs.spatial_models.discrete_domain.**NDGridDomain**(*dim*, *ijk*, *shape*, *affine*, *local_volume*, *topology*, *referential=''*)

Bases: *StructuredDomain*

Particular instance of StructuredDomain, that receives 3 additional variables: affine: array of shape (dim+1, dim+1),

affine transform that maps points to a coordinate system

shape: dim-tuple,
shape of the domain

ijk: array of shape(size, dim), int
grid coordinates of the points

This is to allow easy conversion to images when dim==3, and for compatibility with previous classes

__init__(*dim*, *ijk*, *shape*, *affine*, *local_volume*, *topology*, *referential=''*)

Initialize ndgrid domain instance

Parameters

dim: int,
the (physical) dimension of the domain

ijk: array of shape(size, dim), int
grid coordinates of the points

shape: dim-tuple,
shape of the domain

affine: array of shape (dim+1, dim+1),
affine transform that maps points to a coordinate system

local_volume: array of shape(size),
yields the volume associated with each site

topology: sparse binary coo_matrix of shape (size, size),
that yields the neighboring locations in the domain

referential: string, optional,
identifier of the referential of the coordinates system

Notes

FIXME: local_volume might be computed on-the-fly as **|det(affine)|**

`connected_components()`

returns a labelling of the domain into connected components

`copy()`

Returns a copy of self

`get_coord()`

Returns self.coord

`get_feature(`*fid*`)`

Return self.features[fid]

`get_volume()`

Returns self.local_volume

`integrate(`*fid*`)`

Integrate certain feature over the domain and returns the result

Parameters

fid
[string, feature identifier,] by default, the 1 function is integrataed, yielding domain volume

Returns

lsum = array of shape (self.feature[fid].shape[1]),
the result

`make_feature_from_image(`*path, fid=''*`)`

Extract the information from an image to make it a domain a feature

Parameters

path: string or Nifti1Image instance,
the image from which one wished to extract data

fid: string, optional
identifier of the resulting feature. if ‘’, the feature is not stored

Returns

the corresponding set of values

`mask`(*bmask*)

Returns an instance of self that has been further masked

`representative_feature`(*fid*, *method*)

Compute a statistical representative of the within-Foain feature

Parameters

fid: string, feature id

method: string, method used to compute a representative
to be chosen among 'mean', 'max', 'median', 'min'

`set_feature`(*fid*, *data*, *override=True*)

Append a feature 'fid'

Parameters

fid: string,
feature identifier

data: array of shape(self.size, p) or self.size
the feature data

`to_image`(*path=None*, *data=None*)

Write itself as a binary image, and returns it

Parameters

path: string, path of the output image, if any

data: array of shape self.size,
data to put in the nonzer-region of the image

107.2.4 StructuredDomain

class nipy.labs.spatial_models.discrete_domain.**StructuredDomain**(*dim*, *coord*, *local_volume*, *topology*, *did=''*, *referential=''*)

Bases: *DiscreteDomain*

Besides DiscreteDomain attributed, StructuredDomain has a topology, which allows many operations (morphology etc.)

`__init__`(*dim*, *coord*, *local_volume*, *topology*, *did=''*, *referential=''*)

Initialize structured domain instance

Parameters

dim: int,
the (physical) dimension of the domain

coord: array of shape(size, em_dim),
explicit coordinates of the domain sites

local_volume: array of shape(size),
yields the volume associated with each site

topology: sparse binary coo_matrix of shape (size, size),
that yields the neighboring locations in the domain

did: string, optional,
domain identifier

referential: string, optional,
identifier of the referential of the coordinates system

`connected_components()`

returns a labelling of the domain into connected components

`copy()`

Returns a copy of self

`get_coord()`

Returns self.coord

`get_feature(`*fid*`)`

Return self.features[fid]

`get_volume()`

Returns self.local_volume

`integrate(`*fid*`)`

Integrate certain feature over the domain and returns the result

Parameters

fid
[string, feature identifier,] by default, the 1 function is integrataed, yielding domain volume

Returns

lsum = array of shape (self.feature[fid].shape[1]),
the result

`mask(`*bmask*, *did=''*`)`

Returns a StructuredDomain instance that has been further masked

`representative_feature(`*fid*, *method*`)`

Compute a statistical representative of the within-Foain feature

Parameters

fid: string, feature id
method: string, method used to compute a representative
to be chosen among 'mean', 'max', 'median', 'min'

`set_feature(`*fid*, *data*, *override=True*`)`

Append a feature 'fid'

Parameters

fid: string,
feature identifier

data: array of shape(self.size, p) or self.size
the feature data

107.3 Functions

nipy.labs.spatial_models.discrete_domain.**array_affine_coord**(*mask*, *affine*)

Compute coordinates from a boolean array and an affine transform

Parameters

mask: nd array,
input array, interpreted as a mask

affine: (n+1, n+1) matrix,
affine transform that maps the mask points to some embedding space

Returns

coords: array of shape(sum(mask>0), n),
the computed coordinates

nipy.labs.spatial_models.discrete_domain.**domain_from_binary_array**(*mask*, *affine=None*, *nn=0*)

Return a StructuredDomain from an n-d array

Parameters

mask: np.array instance
a supposedly boolean array that represents the domain

affine: np.array, optional
affine transform that maps the array coordinates to some embedding space by default, this is np.eye(dim+1, dim+1)

nn: neighboring system considered
unused at the moment

nipy.labs.spatial_models.discrete_domain.**domain_from_image**(*mim*, *nn=18*)

Return a StructuredDomain instance from the input mask image

Parameters

mim: NiftiIImage instance, or string path toward such an image
supposedly a mask (where is used to create the DD)

nn: int, optional
neighboring system considered from the image can be 6, 18 or 26

Returns

The corresponding StructuredDomain instance

nipy.labs.spatial_models.discrete_domain.**domain_from_mesh**(*mesh*)

Instantiate a StructuredDomain from a gifti mesh

Parameters

mesh: nibabel gifti mesh instance, or path to such a mesh

nipy.labs.spatial_models.discrete_domain.**grid_domain_from_binary_array**(*mask*, *affine=None*, *nn=0*)

Return a NDGridDomain from an n-d array

Parameters

mask: np.array instance
a supposedly boolean array that represents the domain

affine: np.array, optional
affine transform that maps the array coordinates to some embedding space by default, this is np.eye(dim+1, dim+1)

nn: neighboring system considered
unused at the moment

`nipy.labs.spatial_models.discrete_domain.`**`grid_domain_from_image`**`(`*mim*, *nn=18*`)`

Return a NDGridDomain instance from the input mask image

Parameters

mim: NiftiIImage instance, or string path toward such an image
supposedly a mask (where is used to create the DD)

nn: int, optional
neighboring system considered from the image can be 6, 18 or 26

Returns

The corresponding NDGridDomain instance

`nipy.labs.spatial_models.discrete_domain.`**`grid_domain_from_shape`**`(`*shape*, *affine=None*`)`

Return a NDGridDomain from an n-d array

Parameters

shape: tuple
the shape of a rectangular domain.

affine: np.array, optional
affine transform that maps the array coordinates to some embedding space. By default, this is np.eye(dim+1, dim+1)

`nipy.labs.spatial_models.discrete_domain.`**`idx_affine_coord`**`(`*idx*, *affine*`)`

Compute coordinates from a set of indexes and an affine transform

Parameters

idx:array of shape (n_samples, dim), type int
indexes of certain positions in a nd space

affine: (n+1, n+1) matrix,
affine transform that maps the mask points to some embedding space

Returns

coords: array of shape(sum(mask>0), n),
the computed coordinates

`nipy.labs.spatial_models.discrete_domain.`**`reduce_coo_matrix`**`(`*mat*, *mask*`)`

Reduce a supposedly coo_matrix to the vertices in the mask

Parameters

mat: sparse coo_matrix,
input matrix

mask: boolean array of shape mat.shape[0],
desired elements

`nipy.labs.spatial_models.discrete_domain.`**`smatrix_from_3d_array`**`(`*mask*, *nn=18*`)`

Create a sparse adjacency matrix from an array

Parameters

mask
[3d array,] input array, interpreted as a mask

nn: int, optional
3d neighboring system to be chosen within {6, 18, 26}

Returns

coo_mat: a sparse coo matrix,
adjacency of the neighboring system

`nipy.labs.spatial_models.discrete_domain.`**`smatrix_from_3d_idx`**(*ijk*, *nn=18*)

Create a sparse adjacency matrix from 3d index system

Parameters

idx:array of shape (n_samples, 3), type int
indexes of certain positions in a 3d space

nn: int, optional
3d neighboring system to be chosen within {6, 18, 26}

Returns

coo_mat: a sparse coo matrix,
adjacency of the neighboring system

`nipy.labs.spatial_models.discrete_domain.`**`smatrix_from_nd_array`**(*mask*, *nn=0*)

Create a sparse adjacency matrix from an arbitrary nd array

Parameters

mask
[nd array,] input array, interpreted as a mask

nn: int, optional
nd neighboring system, unused at the moment

Returns

coo_mat: a sparse coo matrix,
adjacency of the neighboring system

`nipy.labs.spatial_models.discrete_domain.`**`smatrix_from_nd_idx`**(*idx*, *nn=0*)

Create a sparse adjacency matrix from nd index system

Parameters

idx:array of shape (n_samples, dim), type int
indexes of certain positions in a nd space

nn: int, optional
nd neighboring system, unused at the moment

Returns

coo_mat: a sparse coo matrix,
adjacency of the neighboring system

CHAPTER

EIGHT

LABS.SPATIAL_MODELS.HIERARCHICAL_PARCELLATION

108.1 Module: `labs.spatial_models.hierarchical_parcellation`

Computation of parcellations using a hierarchical approach. Author: Bertrand Thirion, 2008

108.2 Functions

`nipy.labs.spatial_models.hierarchical_parcellation.`**`hparcel`**(*domain*, *ldata*, *nb_parcel*, *nb_perm=0*, *niter=5*, *mu=10.0*, *dmax=10.0*, *lamb=100.0*, *chunksize=100000.0*, *verbose=0*, *initial_mask=None*)

Function that performs the parcellation by optimizing the inter-subject similarity while retaining the connectedness within subject and some consistency across subjects.

Parameters

domain: discrete_domain.DiscreteDomain instance,
yields all the spatial information on the parcelled domain

ldata: list of (n_subj) arrays of shape (domain.size, dim)
the feature data used to inform the parcellation

nb_parcel: int,
the number of parcels

nb_perm: int, optional,
the number of times the parcellation and prfx computation is performed on sign-swaped data

niter: int, optional,
number of iterations to obtain the convergence of the method information in the clustering algorithm

mu: float, optional,
relative weight of anatomical information

dmax: float optional,
radius of allowed deformations

lamb: float optional
parameter to control the relative importance of space vs function

chunksize; int, optional
number of points used in internal sub-sampling

verbose: bool, optional,
verbosity mode

initial_mask: array of shape (domain.size, nb_subj), optional
initial subject-depedent masking of the domain

Returns

Pa: the resulting parcellation structure appended with the labelling

`nipy.labs.spatial_models.hierarchical_parcellation.`**`perm_prfx`**(*domain*, *graphs*, *features*, *nb_parcel*, *ldata*, *initial_mask=None*, *nb_perm=100*, *niter=5*, *dmax=10.0*, *lamb=100.0*, *chunksize=100000.0*, *verbose=1*)

caveat: assumes that the functional dimension is 1

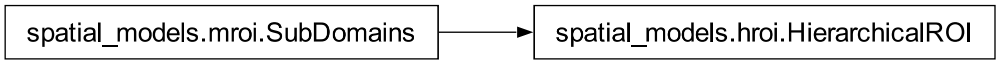
CHAPTER

NINE

LABS.SPATIAL_MODELS.HROI

109.1 Module: `labs.spatial_models.hroi`

Inheritance diagram for `nipy.labs.spatial_models.hroi`:


spatial_models.mroi.SubDomains
spatial_models.hroi.HierarchicalROI

This module contains the specification of 'hierarchical ROI' object, Which is used in spatial models of the library such as structural analysis

The connection with other classes is not completely satisfactory at the moment: there should be some intermediate classes between 'Fields' and 'hroi'

Author

[Bertrand Thirion, 2009-2011] Virgile Fritsch <virgile.fritsch@inria.fr>

109.2 Class

109.3 `HierarchicalROI`

class `nipy.labs.spatial_models.hroi.`**`HierarchicalROI`**(*domain*, *label*, *parents*, *id=None*)

Bases: *`SubDomains`*

Class that handles hierarchical ROIs

Parameters

k
[int] Number of ROI in the SubDomains object

label
[array of shape (domain.size), dtype=np.int_] An array use to define which voxel belongs to which ROI. The label values greater than -1 correspond to subregions labelling. The labels are recomputed so as to be consecutive integers. The labels should not be accessed outside this class. One has to use the API mapping methods instead.

features
[dict {str: list of object, length=self.k}] Describe the voxels features, grouped by ROI

roi_features
[dict {str: array-like, shape=(self.k, roi_feature_dim)] Describe the ROI features. A special feature, *id*, is read-only and is used to give an unique identifier for region, which is persistent through the MROI objects manipulations. On should access the different ROI's features using ids.

parents
[np.ndarray, shape(self.k)] self.parents[i] is the index of the parent of the i-th ROI.

TODO: have the parents as a list of id rather than a list of indices.

`__init__`(*domain*, *label*, *parents*, *id=None*)

Building the HierarchicalROI

`copy`()

Returns a copy of self.

self.domain is not copied.

`feature_to_voxel_map`(*fid*, *roi=False*, *method='mean'*)

Convert a feature to a flat voxel-mapping array.

Parameters

fid: str
Identifier of the feature to be mapped.

roi: bool, optional
If True, compute the map from a ROI feature.

method: str, optional
Representative feature computation method if *fid* is a feature and *roi* is True.

Returns

res: array-like, shape=(domain.size, feature_dim)
A flat array, giving the correspondence between voxels and the feature.

`get_coord`(*id=None*)

Get coordinates of ROI's voxels

Parameters

id: any hashable type
Id of the ROI from which we want the voxels' coordinates. Can be None (default) if we want all ROIs's voxels coordinates.

Returns

coords: array-like, shape=(roi_size, domain_dimension)

if an id is provided,
or list of arrays of shape(roi_size, domain_dimension)

if no id provided (default)

`get_feature`(*fid*, *id=None*)

Return a voxel-wise feature, grouped by ROI.

Parameters

fid: str,
Feature to be returned

id: any hashable type
Id of the ROI from which we want to get the feature. Can be None (default) if we want all ROIs's features.

Returns

feature: array-like, shape=(roi_size, feature_dim)

if an id is provided,
or list of arrays, shape=(roi_size, feature_dim)

if no id provided (default)

`get_id()`

Return ROI's id list.

Users must access ROIs with the use of the identifiers of this list and the methods that give access to their properties/features.

`get_leaves_id()`

Return the ids of the leaves.

`get_local_volume`(*id=None*)

Get volume of ROI's voxels

Parameters

id: any hashable type
Id of the ROI from which we want the voxels' volumes. Can be None (default) if we want all ROIs's voxels volumes.

Returns

loc_volume: array-like, shape=(roi_size,),

if an id is provided,
or list of arrays of shape(roi_size,)

if no id provided (default)

`get_parents()`

Return the parent of each node in the hierarchy

The parents are represented by their position in the nodes flat list.

TODO: The purpose of this class API is not to rely on this order, so we should have self.parents as a list of ids instead of a list of positions

`get_roi_feature`(*fid*, *id=None*)

`get_size`(*id=None*, *ignore_children=True*)

Get ROI size (counted in terms of voxels)

Parameters

id: any hashable type, optional
Id of the ROI from which we want to get the size. Can be None (default) if we want all ROIs's sizes.

ignore_children: bool, optional
Specify if the size of the node should include (ignore_children = False) or not the one of its children (ignore_children = True).

Returns

size: int

if an id is provided,
or list of int

if no id provided (default)

`get_volume`(*id=None*, *ignore_children=True*)

Get ROI volume

Parameters

id: any hashable type, optional
Id of the ROI from which we want to get the volume. Can be None (default) if we want all ROIs's volumes.

ignore_children
[bool, optional] Specify if the volume of the node should include (ignore_children = False) or not the one of its children (ignore_children = True).

Returns

volume
[float]

if an id is provided,
or list of float

if no id provided (default)

`integrate`(*fid=None*, *id=None*)

Integrate certain feature on each ROI and return the k results

Parameters

fid
[str] Feature identifier. By default, the 1 function is integrated, yielding ROI volumes.

id: any hashable type
The ROI on which we want to integrate. Can be None if we want the results for every region.

Returns

lsum = array of shape (self.k, self.feature[fid].shape[1]),
The results

`make_forest`()

Output an nipy forest structure to represent the ROI hierarchy.

`make_graph`()

Output an nipy graph structure to represent the ROI hierarchy.

`merge_ascending`(*id_list*, *pull_features=None*)

Remove the non-valid ROIs by including them in their parents when it exists.

Parameters

id_list: list of id (any hashable type)
The id of the ROI to be merged into their parents. Nodes that are their own parent are unmodified.

pull_features: list of str
List of the ROI features that will be pooled from the children when they are merged into their parents. Otherwise, the receiving parent would keep its own ROI feature.

`merge_descending`(*pull_features=None*)

Remove the items with only one son by including them in their son

Parameters

methods indicates the way possible features are dealt with
(not implemented yet)

Notes

Caveat: if roi_features have been defined, they will be removed

`plot_feature`(*fid*, *ax=None*)

Boxplot the distribution of features within ROIs. Note that this assumes 1-d features.

Parameters

fid: string
the feature identifier

ax: axis handle, optional

`recompute_labels`()

Redefine labels so that they are consecutive integers.

Labels are used as a map to associate voxels to a given ROI. It is an inner object that should not be accessed outside this class. The number of nodes is updated appropriately.

Notes

This method must be called every time the MROI structure is modified.

`reduce_to_leaves`()

Create a new set of rois which are only the leaves of self.

Modification of the structure is done in place. One way therefore want to work on a copy a of a given HROI object.

`remove_feature`(*fid*)

Remove a certain feature

Parameters

fid: str
Feature id

Returns

f
[object] The removed feature.

`remove_roi_feature`(*fid*)

Remove a certain ROI feature.

The *id* ROI feature cannot be removed.

Returns

f
[object] The removed Roi feature.

`representative_feature`(*fid*, *method='mean'*, *id=None*, *ignore_children=True*, *assess_quality=True*)

Compute a ROI representative of a given feature.

Parameters

fid: str,
Feature id

method: str,
Method used to compute a representative. Chosen among 'mean' (default), 'max', 'median', 'min', 'weighted mean'.

id: any hashable type
Id of the ROI from which we want to extract a representative feature. Can be None (default) if we want to get all ROIs's representatives.

ignore_children: bool,
Specify if the volume of the node should include (ignore_children = False) or not the one of its children (ignore_children = True).

assess_quality: bool
If True, a new roi feature is created, which represent the quality of the feature representative (the number of non-nan value for the feature over the ROI size). Default is False.

`select_id`(*id*, *roi=True*)

Convert a ROI id into an index to be used to index features safely.

Parameters

id
[any hashable type, must be in self.get_id()] The id of the region one wants to access.

roi
[bool] If True (default), return the ROI index in the ROI list. If False, return the indices of the voxels of the ROI with the given id. That way, internal access to self.label can be made.

Returns

index
[int or np.array of shape (roi.size,)] Either the position of the ROI in the ROI list (if roi == True), or the positions of the voxels of the ROI with id *id* with respect to the self.label array.

`select_roi`(*id_list*)

Returns an instance of HROI with only the subset of chosen ROIs.

The hierarchy is set accordingly.

Parameters

id_list: list of id (any hashable type)
The id of the ROI to be kept in the structure.

`set_feature`(*fid*, *data*, *id=None*, *override=False*)

Append or modify a feature

Parameters

fid
[str] feature identifier

data: list or array
The feature data. Can be a list of self.k arrays of shape(self.size[k], p) or array of shape(self.size[k])

id: any hashable type, optional
Id of the ROI from which we want to set the feature. Can be None (default) if we want to set all ROIs's features.

override: bool, optional
Allow feature overriding

Note that we cannot create a feature having the same name than a ROI feature.

`set_roi_feature`(*fid*, *data*, *id=None*, *override=False*)

Append or modify a ROI feature

Parameters

fid: str,
feature identifier

data: list of self.k features or a single feature
The ROI feature data

id: any hashable type
Id of the ROI of which we want to set the ROI feature. Can be None (default) if we want to set all ROIs's ROI features.

override: bool, optional,
Allow feature overriding

Note that we cannot create a ROI feature having the same name than a feature.
Note that the `id` feature cannot be modified as an internal component.

`to_image`(*fid=None*, *roi=False*, *method='mean'*, *descrip=None*)

Generates a label image that represents self.

Parameters

fid: str,
Feature to be represented. If None, a binary image of the MROI domain will be we created.

roi: bool,
Whether or not to write the desired feature as a ROI one. (i.e. a ROI feature corresponding to *fid* will be looked upon, and if not found, a representative feature will be computed from the *fid* feature).

method: str,
If a feature is written as a ROI feature, this keyword tweaks the way the representative feature is computed.

descrip: str,
Description of the image, to be written in its header.

Returns

nim
[nibabel nifti image] Nifti image corresponding to the ROI feature to be written.

Notes

Requires that self.dom is an ddom.NDGridDomain

109.4 Functions

`nipy.labs.spatial_models.hroi.`**`HROI_as_discrete_domain_blobs`**(*domain*, *data*, *threshold=-inf*, *smin=0*, *criterion='size'*)

Instantiate an HierarchicalROI as the blob decomposition of data in a certain domain.

Parameters

domain
[discrete_domain.StructuredDomain instance,] Definition of the spatial context.

data
[array of shape (domain.size)] The corresponding data field.

threshold
[float, optional] Thresholding level.

criterion
[string, optional] To be chosen among 'size' or 'volume'.

smin: float, optional
A threshold on the criterion.

Returns

nroi: HierachicalROI instance with a *signal* feature.

`nipy.labs.spatial_models.hroi.`**`HROI_from_watershed`**(*domain*, *data*, *threshold=-inf*)

Instantiate an HierarchicalROI as the watershed of a certain dataset

Parameters

domain: discrete_domain.StructuredDomain instance
Definition of the spatial context.

data: array of shape (domain.size)
The corresponding data field.

threshold: float, optional
Thresholding level.

Returns

nroi
[`HierarchichalROI` instance] The HierachicalROI instance with a `seed` feature.

`nipy.labs.spatial_models.hroi.`**`hroi_agglomeration`**(*input_hroi*, *criterion='size'*, *smin=0*)

Performs an agglomeration then a selection of regions so that a certain size or volume criterion is satisfied.

Parameters

input_hroi: HierarchicalROI instance
The input hROI

criterion: str, optional
To be chosen among 'size' or 'volume'

smin: float, optional
The applied criterion

Returns

output_hroi: HierarchicalROI instance

`nipy.labs.spatial_models.hroi.`**`make_hroi_from_subdomain`**(*sub_domain*, *parents*)

Instantiate an HROi from a SubDomain instance and parents

CHAPTER

LABS.SPATIAL_MODELS.MROI

110.1 Module: `labs.spatial_models.mroi`

Inheritance diagram for `nipy.labs.spatial_models.mroi`:

110.2 Class

110.3 `SubDomains`

class `nipy.labs.spatial_models.mroi.`**`SubDomains`**(*domain*, *label*, *id=None*)

Bases: `object`

This is a class to represent multiple ROI objects, where the reference to a given domain is explicit.

A multiple ROI object is a set of ROI defined on a given domain, each having its own 'region-level' characteristics (ROI features).

Every voxel of the domain can have its own characteristics yet, defined at the 'voxel-level', but those features can only be accessed familywise (i.e. the values are grouped by ROI).

Parameters

k
[int] Number of ROI in the SubDomains object

label
[array of shape (domain.size), dtype=np.int_] An array use to define which voxel belongs to which ROI. The label values greater than -1 correspond to subregions labelling. The labels are recomputed so as to be consecutive integers. The labels should not be accessed outside this class. One has to use the API mapping methods instead.

features
[dict {str: list of object, length=self.k}] Describe the voxels features, grouped by ROI

roi_features
[dict {str: array-like, shape=(self.k, roi_feature_dim)] Describe the ROI features. A special feature, *id*, is read-only and is used to give an unique identifier for region, which is persistent through the MROI objects manipulations. On should access the different ROI's features using ids.

`__init__`(*domain*, *label*, *id=None*)

Initialize subdomains instance

Parameters

domain: ROI instance
defines the spatial context of the SubDomains

label: array of shape (domain.size), dtype=np.int_,
An array use to define which voxel belongs to which ROI. The label values greater than -1 correspond to subregions labelling. The labels are recomputed so as to be consecutive integers. The labels should not be accessed outside this class. One has to use the select_id() mapping method instead.

id: array of shape (n_roi)
Define the ROI identifiers. Once an id has been associated to a ROI it becomes impossible to change it using the API. Hence, one should access ROI through their id to avoid hazardous manipulations.

`copy`()

Returns a copy of self.

Note that self.domain is not copied.

`feature_to_voxel_map`(*fid*, *roi=False*, *method='mean'*)

Convert a feature to a flat voxel-mapping array.

Parameters

fid: str
Identifier of the feature to be mapped.

roi: bool, optional
If True, compute the map from a ROI feature.

method: str, optional
Representative feature computation method if *fid* is a feature and *roi* is True.

Returns

res: array-like, shape=(domain.size, feature_dim)
A flat array, giving the correspondence between voxels and the feature.

`get_coord`(*id=None*)

Get coordinates of ROI's voxels

Parameters

id: any hashable type
Id of the ROI from which we want the voxels' coordinates. Can be None (default) if we want all ROIs's voxels coordinates.

Returns

coords: array-like, shape=(roi_size, domain_dimension)

if an id is provided,
or list of arrays of shape(roi_size, domain_dimension)

if no id provided (default)

`get_feature`(*fid*, *id=None*)

Return a voxel-wise feature, grouped by ROI.

Parameters

fid: str,
Feature to be returned

id: any hashable type
Id of the ROI from which we want to get the feature. Can be None (default) if we want all ROIs's features.

Returns

feature: array-like, shape=(roi_size, feature_dim)

if an id is provided,
or list of arrays, shape=(roi_size, feature_dim)

if no id provided (default)

`get_id`()

Return ROI's id list.

Users must access ROIs with the use of the identifiers of this list and the methods that give access to their properties/features.

`get_local_volume`(*id=None*)

Get volume of ROI's voxels

Parameters

id: any hashable type
Id of the ROI from which we want the voxels' volumes. Can be None (default) if we want all ROIs's voxels volumes.

Returns

loc_volume: array-like, shape=(roi_size,),

if an id is provided,
or list of arrays of shape(roi_size,)

if no id provided (default)

`get_roi_feature`(*fid*, *id=None*)

`get_size`(*id=None*)

Get ROI size (counted in terms of voxels)

Parameters

id: any hashable type
Id of the ROI from which we want to get the size. Can be None (default) if we want all ROIs's sizes.

Returns

size: int

if an id is provided,
or list of int

if no id provided (default)

`get_volume`(*id=None*)

Get ROI volume

Parameters

id: any hashable type
Id of the ROI from which we want to get the volume. Can be None (default) if we want all ROIs's volumes.

Returns

volume
[float]

if an id is provided,
or list of float

if no id provided (default)

`integrate`(*fid=None*, *id=None*)

Integrate certain feature on each ROI and return the k results

Parameters

fid
[str] Feature identifier. By default, the 1 function is integrated, yielding ROI volumes.

id: any hashable type
The ROI on which we want to integrate. Can be None if we want the results for every region.

Returns

lsum = array of shape (self.k, self.feature[fid].shape[1]),
The results

`plot_feature`(*fid*, *ax=None*)

Boxplot the distribution of features within ROIs. Note that this assumes 1-d features.

Parameters

fid: string
the feature identifier

ax: axis handle, optional

`recompute_labels`()

Redefine labels so that they are consecutive integers.

Labels are used as a map to associate voxels to a given ROI. It is an inner object that should not be accessed outside this class. The number of nodes is updated appropriately.

Notes

This method must be called every time the MROI structure is modified.

`remove_feature`(*fid*)

Remove a certain feature

Parameters

fid: str
Feature id

Returns

f
[object] The removed feature.

`remove_roi_feature`(*fid*)

Remove a certain ROI feature.

The *id* ROI feature cannot be removed.

Returns

f
[object] The removed Roi feature.

`representative_feature`(*fid*, *method='mean'*, *id=None*, *assess_quality=False*)

Compute a ROI representative of a given feature.

Parameters

fid
[str] Feature id

method
[str, optional] Method used to compute a representative. Chosen among 'mean' (default), 'max', 'median', 'min', 'weighted mean'.

id
[any hashable type, optional] Id of the ROI from which we want to extract a representative feature. Can be None (default) if we want to get all ROIs's representatives.

assess_quality: bool, optional
If True, a new roi feature is created, which represent the quality of the feature representative (the number of non-nan value for the feature over the ROI size). Default is False.

Returns

summary_feature: np.ndarray, shape=(self.k, feature_dim)
Representative feature computed according to *method*.

`select_id`(*id*, *roi=True*)

Convert a ROI id into an index to be used to index features safely.

Parameters

id
[any hashable type, must be in self.get_id()] The id of the region one wants to access.

roi
[bool] If True (default), return the ROI index in the ROI list. If False, return the indices of the voxels of the ROI with the given id. That way, internal access to self.label can be made.

Returns

index
[int or np.array of shape (roi.size,)] Either the position of the ROI in the ROI list (if roi == True), or the positions of the voxels of the ROI with id *id* with respect to the self.label array.

`select_roi`(*id_list*)

Returns an instance of MROI with only the subset of chosen ROIs.

Parameters

id_list: list of id (any hashable type)
The id of the ROI to be kept in the structure.

`set_feature`(*fid*, *data*, *id=None*, *override=False*)

Append or modify a feature

Parameters

fid
[str] feature identifier

data: list or array
The feature data. Can be a list of self.k arrays of shape(self.size[k], p) or array of shape(self.size[k])

id: any hashable type, optional
Id of the ROI from which we want to set the feature. Can be None (default) if we want to set all ROIs's features.

override: bool, optional
Allow feature overriding

Note that we cannot create a feature having the same name than
a ROI feature.

`set_roi_feature`(*fid*, *data*, *id=None*, *override=False*)

Append or modify a ROI feature

Parameters

fid: str,
feature identifier

data: list of self.k features or a single feature
The ROI feature data

id: any hashable type
Id of the ROI of which we want to set the ROI feature. Can be None (default) if we want to set all ROIs's ROI features.

override: bool, optional,
Allow feature overriding

Note that we cannot create a ROI feature having the same name than
a feature.
Note that the `id` feature cannot be modified as an internal
component.

`to_image`(*fid=None*, *roi=False*, *method='mean'*, *descrip=None*)

Generates a label image that represents self.

Parameters

fid: str,
Feature to be represented. If None, a binary image of the MROI domain will be we created.

roi: bool,
Whether or not to write the desired feature as a ROI one. (i.e. a ROI feature corresponding to *fid* will be looked upon, and if not found, a representative feature will be computed from the *fid* feature).

method: str,
If a feature is written as a ROI feature, this keyword tweaks the way the representative feature is computed.

descrip: str,
Description of the image, to be written in its header.

Returns

nim
[nibabel nifti image] Nifti image corresponding to the ROI feature to be written.

Notes

Requires that self.dom is an ddom.NDGridDomain

110.4 Functions

`nipy.labs.spatial_models.mroi.`**`subdomain_from_array`**(*labels*, *affine=None*, *nn=0*)

Return a SubDomain from an n-d int array

Parameters

label: np.array instance
A supposedly boolean array that yields the regions.

affine: np.array, optional
Affine transform that maps the array coordinates to some embedding space by default, this is np.eye(dim+1, dim+1).

nn: int,
Neighboring system considered. Unused at the moment.

Notes

Only labels > -1 are considered.

`nipy.labs.spatial_models.mroi.`**`subdomain_from_balls`**(*domain*, *positions*, *radii*)

Create discrete ROIs as a set of balls within a certain coordinate systems.

Parameters

domain: StructuredDomain instance,
the description of a discrete domain

positions: array of shape(k, dim):
the positions of the balls

radii: array of shape(k):
the sphere radii

`nipy.labs.spatial_models.mroi.`**`subdomain_from_image`**`(`*mim*, *nn=18*`)`

Return a SubDomain instance from the input mask image.

Parameters

mim: NiftiIImage instance, or string path toward such an image
supposedly a label image

nn: int, optional
Neighboring system considered from the image can be 6, 18 or 26.

Returns

The MultipleROI instance

Notes

Only labels > -1 are considered

`nipy.labs.spatial_models.mroi.`**`subdomain_from_position_and_image`**`(`*nim*, *pos*`)`

Keep the set of labels of the image corresponding to a certain index so that their position is closest to the prescribed one.

Parameters

mim: NiftiIImage instance, or string path toward such an image
supposedly a label image

pos: array of shape(3) or list of length 3,
the prescribed position

CHAPTER

ONE

LABS.SPATIAL_MODELS.PARCEL_IO

111.1 Module: `labs.spatial_models.parcel_io`

Utility functions for mutli-subjectParcellation: this basically uses nipy io lib to perform IO opermation in parcel definition processes

111.2 Functions

`nipy.labs.spatial_models.parcel_io.`**`fixed_parcellation`**(*mask_image*, *betas*, *nbparcel*, *nn=6*, *method='ward'*, *write_dir=None*, *mu=10.0*, *verbose=0*, *fullpath=None*)

Fixed parcellation of a given dataset

Parameters

domain/mask_image

betas: list of paths to activation images from the subject

nbparcel, int
[number of desired parcels]

nn=6: number of nearest neighbors to define the image topology
(6, 18 or 26)

method='ward': clustering method used, to be chosen among
'ward', 'gkm', 'ward_and-gkm' 'ward': Ward's clustering algorithm 'gkm': Geodesic k-means algorithm, random initialization 'gkm_and_ward': idem, initialized by Ward's clustering

write_di: string, topional, write directory.
If fullpath is None too, then no file output.

mu = 10., float: the relative weight of anatomical information

verbose=0: verbosity mode

fullpath=None, string,
path of the output image If write_dir and fullpath are None then no file output. If only fullpath is None then it is the write dir + a name depending on the method.

Notes

Ward's method takes time (about 6 minutes for a 60K voxels dataset)

Geodesic k-means is 'quick and dirty'

Ward's + GKM is expensive but quite good

To reduce CPU time, rather use nn=6 (especially with Ward)

`nipy.labs.spatial_models.parcel_io.`**`mask_parcellation`**`(`*mask_images*, *nb_parcel*, *threshold=0*, *output_image=None*`)`

Performs the parcellation of a certain mask

Parameters

mask_images: string or Nifti1Image or list of strings/Nifti1Images,
paths of mask image(s) that define(s) the common space.

nb_parcel: int,
number of desired parcels

threshold: float, optional,
level of intersection of the masks

output_image: string, optional
path of the output image

Returns

wim: Nifti1Imagine instance, representing the resulting parcellation

`nipy.labs.spatial_models.parcel_io.`**`parcel_input`**`(`*mask_images*, *learning_images*, *ths=0.5*, *fdim=None*`)`
Instantiating a Parcel structure from a give set of input

Parameters

mask_images: string or Nifti1Image or list of strings/Nifti1Images,
paths of mask image(s) that define(s) the common space.

learning_images: (nb_subject-) list of (nb_feature-) list of strings,
paths of feature images used as input to the parcellation procedure

ths=.5: threshold to select the regions that are common across subjects.
if ths = .5, thethreshold is half the number of subjects

fdim: int, optional
if nb_feature (the dimension of the data) used in subsequent analyses if greater than fdim, a PCA is performed to reduce the information in the data Byd efault, no reduction is performed

Returns

domain
[discrete_domain.DiscreteDomain instance] that stores the spatial information on the parcelled domain

feature: (nb_subect-) list of arrays of shape (domain.size, fdim)
feature information available to parcellate the data

`nipy.labs.spatial_models.parcel_io.`**`parcellation_based_analysis`**`(`*Pa*, *test_images*, *test_id='one_sample'*, *rfx_path=None*, *condition_id=''*, *swd=None*`)`

This function computes parcel averages and RFX at the parcel-level

Parameters

Pa: MultiSubjectParcellation instance
the description of the parcellation

test_images: (Pa.nb_subj-) list of paths
paths of images used in the inference procedure

test_id: string, optional,
if test_id=='one_sample', the one_sample statstic is computed otherwise, the parcel-based signal averages are returned

rfx_path: string optional,
path of the resulting one-sample test image, if applicable

swd: string, optional
output directory used to compute output path if rfx_path is not given

condition_id: string, optional,
contrast/condition id used to compute output path

Returns

test_data: array of shape(Pa.nb_parcel, Pa.nb_subj)
the parcel-level signal average if test is not 'one_sample'

prfx: array of shape(Pa.nb_parcel),
the one-sample t-value if test_id is 'one_sample'

`nipy.labs.spatial_models.parcel_io.`**`write_parcellation_images`**(*Pa*, *template_path=None*, *indiv_path=None*, *subject_id=None*, *swd=None*)

Write images that describe the spatial structure of the parcellation

Parameters

Pa
[MultiSubjectParcellation instance,] the description of the parcellation

template_path: string, optional,
path of the group-level parcellation image

indiv_path: list of strings, optional
paths of the individual parcellation images

subject_id: list of strings of length Pa.nb_subj
subject identifiers, used to infer the paths when not available

swd: string, optional
output directory used to infer the paths when these are not available

CHAPTER

TWO

LABS.SPATIAL_MODELS.PARCELLATION

112.1 Module: `labs.spatial_models.parcellation`

Inheritance diagram for `nipy.labs.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

112.2 `MultiSubjectParcellation`

class `nipy.labs.spatial_models.parcellation.`**`MultiSubjectParcellation`**(*domain*, *template_labels=None*, *individual_labels=None*, *nb_parcel=None*)

Bases: `object`

MultiSubjectParcellation class are used to represent parcels that can have different spatial different contours in a given group of subject It consists of self.domain: the specification of a domain self.template_labels the specification of a template parcellation self.individual_labels the specification of individual parcellations

fixme:should inherit from mroi.MultiROI

`__init__`(*domain*, *template_labels=None*, *individual_labels=None*, *nb_parcel=None*)

Initialize multi-subject parcellation

Parameters

domain: discrete_domain.DiscreteDomain instance,
definition of the space considered in the parcellation

template_labels: array of shape domain.size, optional
definition of the template labelling

individual_labels: array of shape (domain.size, nb_subjects), optional,
the individual parcellations corresponding to the template

nb_parcel: int, optional,
number of parcels in the model can be inferred as template_labels.max()+1, or 1 by default cannot be smaller than template_labels.max()+1

`check()`

Performs an elementary check on self

`copy()`

Returns a copy of self

`get_feature(`*fid*`)`

Get feature defined by *fid*

Parameters

fid: string, the feature identifier

`make_feature(`*fid*, *data*`)`

Compute parcel-level averages of data

Parameters

fid: string, the feature identifier
data: array of shape (self.domain.size, self.nb_subj, dim) or
(self.domain.sire, self.nb_subj) Some information at the voxel level

Returns

pfeature: array of shape(self.nb_parcel, self.nbsubj, dim)
the computed feature data

`population()`

Returns the counting of labels per voxel per subject

Returns

population: array of shape (self.nb_parcel, self.nb_subj)

`set_feature(`*fid*, *data*`)`

Set feature defined by *fid* and *data* into `self`

Parameters

fid: string
the feature identifier

data: array of shape (self.nb_parcel, self.nb_subj, dim) or

(self.nb_parcel, self.nb_subj)

the data to be set as parcel- and subject-level information

`set_individual_labels(`*individual_labels*`)`

`set_template_labels(`*template_labels*`)`

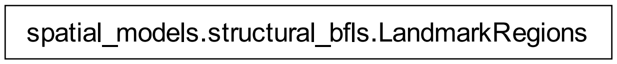
CHAPTER

THREE

LABS.SPATIAL_MODELS.STRUCTURAL_BFLS

113.1 Module: `labs.spatial_models.structural_bfls`

Inheritance diagram for `nipy.labs.spatial_models.structural_bfls`:



The main routine of this module implement the LandmarkRegions class, that is used to represent Regions of interest at the population level (in a template space).

This has been used 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, 2006-2013

113.2 `LandmarkRegions`

class `nipy.labs.spatial_models.structural_bfls.`**`LandmarkRegions`**(*domain*, *k*, *indiv_coord*, *subjects*, *confidence*)

Bases: `object`

This class is intended to represent a set of inter-subject regions It should inherit from some abstract multiple ROI class, not implemented yet.

`__init__`(*domain*, *k*, *indiv_coord*, *subjects*, *confidence*)

Building the landmark_region

Parameters

domain: ROI instance
defines the spatial context of the SubDomains

k: int,
the number of landmark regions considered

indiv_coord: k-length list of arrays,
coordinates of the nodes in some embedding space.

subjects: k-length list of integers
these correspond to an ROI feature: the subject index of individual regions

confidence: k-length list of arrays,
confidence values for the regions (0 is low, 1 is high)

`centers()`

returns the average of the coordinates for each region

`kernel_density`(*k=None*, *coord=None*, *sigma=1.0*)

Compute the density of a component as a kde

Parameters

k: int (<= self.k) or None
component upon which the density is computed if None, the sum is taken over k

coord: array of shape(n, self.dom.em_dim), optional
a set of input coordinates

sigma: float, optional
kernel size

Returns

kde: array of shape(n)
the density sampled at the coords

`map_label`(*coord=None*, *pval=1.0*, *sigma=1.0*)

Sample the set of landmark regions on the proposed coordinate set cs, assuming a Gaussian shape

Parameters

coord: array of shape(n,dim), optional,
a set of input coordinates

pval: float in [0,1]), optional
cutoff for the CR, i.e. highest posterior density threshold

sigma: float, positive, optional
spatial scale of the spatial model

Returns

label: array of shape (n): the posterior labelling

`roi_prevalence()`

Return a confidence index over the different rois

Returns

confid: array of shape self.k
the population_prevalence

`show()`

function to print basic information on self

`nipy.labs.spatial_models.structural_bfls.build_landmarks`(*domain*, *coords*, *subjects*, *labels*, *confidence=None*, *prevalence_pval=0.95*, *prevalence_threshold=0*, *sigma=1.0*)

Given a list of hierarchical ROIs, and an associated labelling, this creates an Amer structure wuch groups ROIs with the same label.

Parameters

domain: discrete_domain.DiscreteDomain instance,
description of the spatial context of the landmarks

coords: array of shape(n, 3)
Sets of coordinates for the different objects

subjects: array of shape (n), dtype = np.int_
indicators of the dataset the objects come from

labels: array of shape (n), dtype = np.int_
index of the landmark the object is associated with

confidence: array of shape (n),
measure of the significance of the regions

prevalence_pval: float, optional
prevalence_threshold: float, optional,
(c) A label should be present in prevalence_threshold subjects with a probability>prevalence_pval in order to be valid

sigma: float optional,
regularizing constant that defines a prior on the region extent

Returns

LR
[None or structural_bfls.LR instance] describing a cross-subject set of ROIs. If inference yields a null result, LR is set to None

newlabel: array of shape (n)
a relabelling of the individual ROIs, similar to u, that discards labels that do not fulfill the condition (c)

CHAPTER FOUR

LABS.STATISTICAL_MAPPING

114.1 Module: `labs.statistical_mapping`

Inheritance diagram for `nipy.labs.statistical_mapping`:

114.2 Class

114.3 `LinearModel`

class `nipy.labs.statistical_mapping.`**`LinearModel`**(*data*, *design_matrix*, *mask=None*, *formula=None*, *model='spherical'*, *method=None*, *niter=2*)

Bases: `object`

`__init__`(*data*, *design_matrix*, *mask=None*, *formula=None*, *model='spherical'*, *method=None*, *niter=2*)

`contrast`(*vector*)

Compute images of contrast and contrast variance.

`def_model = 'spherical'`

`def_niter = 2`

`dump`(*filename*)

Dump GLM fit as npz file.

114.4 Functions

`nipy.labs.statistical_mapping.`**`bonferroni`**`(`*p*, *n*`)`

`nipy.labs.statistical_mapping.`**`cluster_stats`**`(`*zimg*, *mask*, *height_th*, *height_control='fpr'*, *cluster_th=0*, *nulls={}*`)`

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.

Parameters

zimg: z-score image
mask: mask image
height_th: cluster forming threshold
height_control: string
false positive control meaning of cluster forming threshold: 'fpr'|'fdr'|'bonferroni'|'none'

cluster_th: cluster size threshold
null_s
[cluster-level calibration method: None|'rft'|array]

Notes

This works only with three dimensional data

`nipy.labs.statistical_mapping.`**`get_3d_peaks`**`(`*image*, *mask=None*, *threshold=0.0*, *nn=18*, *order_th=0*`)`

returns all the peaks of image that are with the mask and above the provided threshold

Parameters

image, (3d) test image
mask=None, (3d) mask image
By default no masking is performed

threshold=0., float, threshold value above which peaks are considered
nn=18, int, number of neighbours of the topological spatial model
order_th=0, int, threshold on topological order to validate the peaks

Returns

peaks, a list of dictionaries, where each dict has the fields:
vals, map value at the peak
order, topological order of the peak
ijk, array of shape (1,3) grid coordinate of the peak
pos, array of shape (n_maxima,3) mm coordinates (mapped by affine)
of the peaks

`nipy.labs.statistical_mapping.`**`linear_model_fit`**`(`*data_images*, *mask_images*, *design_matrix*, *vector*`)`

Helper function for group data analysis using arbitrary design matrix

`nipy.labs.statistical_mapping.`**`onesample_test`**`(`*data_images*, *vardata_images*, *mask_images*, *stat_id*, *permutations=0*, *cluster_forming_th=0.01*`)`

Helper function for permutation-based mass univariate onesample group analysis.

`nipy.labs.statistical_mapping.`**`prepare_arrays`**`(`*data_images*, *vardata_images*, *mask_images*`)`

`nipy.labs.statistical_mapping.`**`simulated_pvalue`**`(`*t*, *simu_t*`)` `nipy.labs.statistical_mapping.`**`twosample_test`**(*data_images*, *vardata_images*, *mask_images*, *labels*, *stat_id*, *permutations=0*, *cluster_forming_th=0.01*)

Helper function for permutation-based mass univariate twosample group analysis. Labels is a binary vector (1-2). Regions more active for group 1 than group 2 are inferred.

CHAPTER

FIVE

LABS.UTILS.REPRODUCIBILITY_MEASURES

115.1 Module: `labs.utils.reproducibility_measures`

Functions for computing reproducibility measures.

General procedure is:

- dataset is subject to jacknife subampling ('splitting'),
- each subsample being analysed independently,
- a reproducibility measure is then derived;

It is used to produce the work described in Analysis of a large fMRI cohort:

Statistical and methodological issues for group analyses. Thirion B, Pinel P, Meriaux S, Roche A, Dehaene S, Poline JB. Neuroimage. 2007 Mar;35(1):105-20.

Bertrand Thirion, 2009-2010

115.2 Functions

`nipy.labs.utils.reproducibility_measures.`**`bootstrap_group`**`(`*nsubj*, *ngroups*`)`

Split the proposed group into redundant subgroups by bootstrap

Parameters

nsubj (int) the number of subjects in the population
ngroups(int) Number of subbgroups to be drawn

Returns

samples: a list of ngroups arrays containing
the indexes of the subjects in each subgroup

`nipy.labs.utils.reproducibility_measures.`**`cluster_reproducibility`**`(`*data*, *vardata*, *domain*, *ngroups*, *sigma*, *method='crfx'*, *swap=False*, *verbose=0*, ***kwargs*`)`

Returns a measure of cluster-level reproducibility of activation patterns (i.e. how far clusters are from each other)

Parameters

data: array of shape (nvox,nsubj)
the input data from which everything is computed

vardata: array of shape (nvox,nsubj)
the variance of the data that is also available

domain: referential- and domain- defining image instance
ngroups (int),
Number of subbgroups to be drawn

sigma (float): parameter that encodes how far far is
threshold (float):
binarization threshold

method='crfx', string to be chosen among 'crfx', 'cmfx' or 'cffx'
inference method under study

swap = False: if True, a random sign swap of the data is performed
This is used to simulate a null hypothesis on the data.

verbose=0
[verbosity mode]

Returns

score (float): the desired cluster-level reproducibility index

`nipy.labs.utils.reproducibility_measures.`**`cluster_threshold`**(*stat_map*, *domain*, *th*, *csize*)

Perform a thresholding of a map at the cluster-level

Parameters

stat_map: array of shape(nbvox)
the input data

domain: Nifti1Image instance,
referential- and domain-defining image

th (float): cluster-forming threshold
csize (int>0): cluster size threshold

Returns

binary array of shape (nvox): the binarized thresholded map

Notes

Should be replaced by a more standard function in the future

`nipy.labs.utils.reproducibility_measures.`**`conjunction`**(*x*, *vx*, *k*)

Returns a conjunction statistic as the sum of the k lowest t-values

Parameters

x: array of shape(nrows, ncols),
effect matrix

vx: array of shape(nrows, ncols),
variance matrix

k: int,
number of subjects in the conjunction

Returns

t array of shape(nrows): conjunction statistic

nipy.labs.utils.reproducibility_measures.**coord_bsa**(*domain*, *betas*, *theta=3.0*, *dmax=5.0*, *ths=0*, *thq=0.5*, *smin=0*, *afname=None*)

main function for performing bsa on a dataset where bsa = nipy.labs.spatial_models.bayesian_structural_analysis

Parameters

domain: image instance,
referential- and domain-defining image

betas: array of shape (nbnodes, subjects),
the multi-subject statistical maps

theta: float, optional
first level threshold

dmax: float>0, optional
expected cluster std in the common space in units of coord

ths: int, >=0), optional
representatitivity threshold

thq: float, optional,
posterior significance threshold should be in [0,1]

smin: int, optional,
minimal size of the regions to validate them

afname: string, optional
path where intermediate results cam be pickled

Returns

afcoord array of shape(number_of_regions,3):
coordinate of the found landmark regions

nipy.labs.utils.reproducibility_measures.**draw_samples**(*nsubj*, *ngroups*, *split_method='default'*)

Draw randomly ngroups sets of samples from [0..nsubj-1]

Parameters

nsubj, int, the total number of items
ngroups, int, the number of desired groups
split_method: string, optional,
to be chosen among 'default', 'bootstrap', 'jacknife' if 'bootstrap', then each group will be nsubj

drawn with repetitions among nsubj

if 'jacknife' the population is divided into
ngroups disjoint equally-sized subgroups

if 'default', 'bootstrap' is used when nsubj < 10 * ngroups
otherwise jacknife is used

Returns

samples, a list of ngroups array that represent the subsets.
fixme
[this should allow variable bootstrap,]

i.e. draw ngroups of groupsize among nsubj

`nipy.labs.utils.reproducibility_measures.`**`fttest`**`(`*x*, *vx*`)`

Assuming that x and vx represent a effect and variance estimates, returns a cumulated ('fixed effects') t-test of the data over each row

Parameters

x: array of shape(nrows, ncols): effect matrix
vx: array of shape(nrows, ncols): variance matrix

Returns

t array of shape(nrows): fixed effect statistics array

`nipy.labs.utils.reproducibility_measures.`**`get_cluster_position_from_thresholded_map`**`(`*stat_map*, *domain*, *thr=3.0*, *csize=10*`)`

the clusters above thr of size greater than csize in 18-connectivity are computed

Parameters

stat_map
[array of shape (nbvox),] map to threshold

mask: Nifti1Image instance,
referential- and domain-defining image

thr: float, optional,
cluster-forming threshold

cisze=10: int
cluster size threshold

Returns

positions array of shape(k,anat_dim):
the cluster positions in physical coordinates where k= number of clusters if no such cluster exists, None is returned

`nipy.labs.utils.reproducibility_measures.`**`get_peak_position_from_thresholded_map`**`(`*stat_map*, *domain*, *threshold*`)`

The peaks above thr in 18-connectivity are computed

Parameters

stat_map: array of shape (nbvox): map to threshold
deomain: referential- and domain-defining image
thr, float: cluster-forming threshold

Returns

positions array of shape(k,anat_dim):
the cluster positions in physical coordinates where k= number of clusters if no such cluster exists, None is returned

`nipy.labs.utils.reproducibility_measures.`**`group_reproducibility_metrics`**(*mask_images*, *contrast_images*, *variance_images*, *thresholds*, *ngroups*, *method*, *cluster_threshold=10*, *number_of_samples=10*, *sigma=6.0*, *do_clusters=True*, *do_voxels=True*, *do_peaks=True*, *swap=False*)

Main function to perform reproducibility analysis, including nifti1 io

Parameters

threshold: list or 1-d array,
the thresholds to be tested

Returns

cluster_rep_results: dictionary,
results of cluster-level reproducibility analysis

voxel_rep_results: dictionary,
results of voxel-level reproducibility analysis

peak_rep_results: dictionary,
results of peak-level reproducibility analysis

`nipy.labs.utils.reproducibility_measures.`**`histo_repro`**(*h*)

Given the histogram h, compute a standardized reproducibility measure

Parameters

h array of shape(xmax+1), the histogram values

Returns

hr, float: the measure

`nipy.labs.utils.reproducibility_measures.`**`map_reproducibility`**(*data*, *vardata*, *domain*, *ngroups*, *method='crfx'*, *swap=False*, *verbose=0*, ***kwargs*)

Return a reproducibility map for the given method

Parameters

data: array of shape (nvox,nsubj)
the input data from which everything is computed

vardata: array of the same size
the corresponding variance information

domain: referential- and domain-defining image
ngroups (int): the size of each subrgoup to be studied
threshold (float): binarization threshold
(makes sense only if method==rfx)

method='crfx', string to be chosen among 'crfx', 'cmfx', 'cffx'
inference method under study

verbose=0
[verbosity mode]

Returns

rmap: array of shape(nvox)
the reproducibility map

`nipy.labs.utils.reproducibility_measures.`**`mfx_ttest`**`(`*x*, *vx*`)`

Idem fttest, but returns a mixed-effects statistic

Parameters

x: array of shape(nrows, ncols): effect matrix
vx: array of shape(nrows, ncols): variance matrix

Returns

t array of shape(nrows): mixed effect statistics array

`nipy.labs.utils.reproducibility_measures.`**`peak_reproducibility`**`(`*data*, *vardata*, *domain*, *ngroups*, *sigma*, *method='crfx'*, *swap=False*, *verbose=0*, ***kwargs*`)`

Return a measure of cluster-level reproducibility of activation patterns (i.e. how far clusters are from each other)

Parameters

data: array of shape (nvox,nsubj)
the input data from which everything is computed

vardata: array of shape (nvox,nsubj)
the variance of the data that is also available

domain: referential- and domain-defining image
ngroups (int),
Number of subbgroups to be drawn

sigma: float, parameter that encodes how far far is
threshold: float, binarization threshold
method: string to be chosen among 'crfx', 'cmfx' or 'cffx',
inference method under study

swap = False: if True, a random sign swap of the data is performed
This is used to simulate a null hypothesis on the data.

verbose=0
[verbosity mode]

Returns

score (float): the desired cluster-level reproducibility index

`nipy.labs.utils.reproducibility_measures.`**`split_group`**`(`*nsubj*, *ngroups*`)`

Split the proposed group into random disjoint subgroups

Parameters

nsubj (int) the number of subjects to be split
ngroups(int) Number of subbgroups to be drawn

Returns

samples: a list of ngroups arrays containing
the indexes of the subjects in each subgroup

nipy.labs.utils.reproducibility_measures.**statistics_from_position**(*target*, *data*, *sigma=1.0*)

Return a number characterizing how close data is from target using a kernel-based statistic

Parameters

target: array of shape(nt,anat_dim) or None
the target positions

data: array of shape(nd,anat_dim) or None
the data position

sigma=1.0 (float), kernel parameter
or a distance that say how good good is

Returns

sensitivity (float): how well the targets are fitted
by the data in [0,1] interval 1 is good 0 is bad

nipy.labs.utils.reproducibility_measures.**ttest**(*x*)

Returns the t-test for each row of the data x

nipy.labs.utils.reproducibility_measures.**voxel_reproducibility**(*data*, *vardata*, *domain*, *ngroups*, *method='crfx'*, *swap=False*, *verbose=0*, ***kwargs*)

return a measure of voxel-level reproducibility of activation patterns

Parameters

data: array of shape (nvox,nsubj)
the input data from which everything is computed

vardata: array of shape (nvox,nsubj)
the corresponding variance information ngroups (int): Number of subbgroups to be drawn

domain: referential- and domain-defining image
ngourps: int,
number of groups to be used in the resampling procedure

method: string, to be chosen among 'crfx', 'cmfx', 'cffx'
inference method under study

verbose: bool, verbosity mode

Returns

kappa (float): the desired reproducibility index

nipy.labs.utils.reproducibility_measures.**voxel_thresholded_ttest**(*x*, *threshold*)

Returns a binary map of the ttest>threshold

CHAPTER

SIX

LABS.UTILS.SIMUL_MULTISUBJECT_FMRI_DATASET

116.1 Module: `labs.utils.simul_multisubject_fmri_dataset`

This module contains a function to produce a dataset which simulates a collection of 2D images This dataset is saved as a 3D image (each slice being a subject) and a 3D array

Author : Bertrand Thirion, 2008-2010

116.2 Functions

`nipy.labs.utils.simul_multisubject_fmri_dataset.`**`surrogate_2d_dataset`**(*n_subj=10*, *shape=(30, 30)*, *sk=1.0*, *noise_level=1.0*, *pos=array([[6, 7], [10, 10], [15, 10]])*, *ampli=array([3, 4, 4])*, *spatial_jitter=1.0*, *signal_jitter=1.0*, *width=5.0*, *width_jitter=0*, *out_text_file=None*, *out_image_file=None*, *seed=False*)

Create surrogate (simulated) 2D activation data with spatial noise

Parameters

n_subj: integer, optional
The number of subjects, ie the number of different maps generated.

shape=(30,30): tuple of integers,
the shape of each image

sk: float, optional
Amount of spatial noise smoothness.

noise_level: float, optional
Amplitude of the spatial noise. amplitude=noise_level)

pos: 2D ndarray of integers, optional
x, y positions of the various simulated activations.

ampli: 1D ndarray of floats, optional
Respective amplitude of each activation

spatial_jitter: float, optional
Random spatial jitter added to the position of each activation, in pixel.

signal_jitter: float, optional
Random amplitude fluctuation for each activation, added to the amplitude specified by *ampli*

width: float or ndarray, optional
Width of the activations

width_jitter: float
Relative width jitter of the blobs

out_text_file: string or None, optional
If not None, the resulting array is saved as a text file with the given file name

out_image_file: string or None, optional
If not None, the resulting is saved as a nifti file with the given file name.

seed=False: int, optional
If seed is not False, the random number generator is initialized at a certain value

Returns

dataset: 3D ndarray
The surrogate activation map, with dimensions `(n_subj,) + shape`

`nipy.labs.utils.simul_multisubject_fmri_dataset.`**`surrogate_3d_dataset`**(*n_subj=1*, *shape=(20, 20, 20)*, *mask=None*, *sk=1.0*, *noise_level=1.0*, *pos=None*, *ampli=None*, *spatial_jitter=1.0*, *signal_jitter=1.0*, *width=5.0*, *out_text_file=None*, *out_image_file=None*, *seed=False*)

Create surrogate (simulated) 3D activation data with spatial noise.

Parameters

n_subj: integer, optional
The number of subjects, ie the number of different maps generated.

shape=(20,20,20): tuple of 3 integers,
the shape of each image

mask=None: Nifti1Image instance,
referential- and mask- defining image (overrides shape)

sk: float, optional
Amount of spatial noise smoothness.

noise_level: float, optional
Amplitude of the spatial noise. amplitude=noise_level)

pos: 2D ndarray of integers, optional
x, y positions of the various simulated activations.

ampli: 1D ndarray of floats, optional
Respective amplitude of each activation

spatial_jitter: float, optional
Random spatial jitter added to the position of each activation, in pixel.

signal_jitter: float, optional
Random amplitude fluctuation for each activation, added to the amplitude specified by ampli

width: float or ndarray, optional
Width of the activations

out_text_file: string or None, optional
If not None, the resulting array is saved as a text file with the given file name

out_image_file: string or None, optional
If not None, the resulting is saved as a nifti file with the given file name.

seed=False: int, optional
If seed is not False, the random number generator is initialized at a certain value

Returns

dataset: 3D ndarray
The surrogate activation map, with dimensions `(n_subj,) + shape`

`nipy.labs.utils.simul_multisubject_fmri_dataset.`**`surrogate_4d_dataset`**(*shape=(20, 20, 20)*, *mask=None*, *n_scans=1*, *n_sess=1*, *dmtx=None*, *sk=1.0*, *noise_level=1.0*, *signal_level=1.0*, *out_image_file=None*, *seed=False*)

Create surrogate (simulated) 3D activation data with spatial noise.

Parameters

shape = (20, 20, 20): tuple of integers,
the shape of each image

mask=None: brifti image instance,
referential- and mask- defining image (overrides shape)

n_scans: int, optional,
number of scans to be simlulated overridden by the design matrix

n_sess: int, optional,
the number of simulated sessions

dmtx: array of shape(n_scans, n_rows),
the design matrix

sk: float, optional
Amount of spatial noise smoothness.

noise_level: float, optional
Amplitude of the spatial noise. amplitude=noise_level)

signal_level: float, optional,
Amplitude of the signal

out_image_file: string or list of strings or None, optional
If not None, the resulting is saved as (set of) nifti file(s) with the given file path(s)

seed=False: int, optional
If seed is not False, the random number generator is initialized at a certain value

Returns

dataset: a list of n_sess ndarray of shape
(shape[0], shape[1], shape[2], n_scans) The surrogate activation map

CHAPTER

SEVEN

LABS.UTILS.ZSCORE

117.1 Module: `labs.utils.zscore`

`nipy.labs.utils.zscore.`**`zscore`**(*pvalue*)

Return the z-score corresponding to a given p-value.

CHAPTER

EIGHT

LABS.VIZ_TOOLS.ACTIVATION_MAPS

118.1 Module: `labs.viz_tools.activation_maps`

Functions to do automatic visualization of activation-like maps.

For 2D-only visualization, only matplotlib is required. For 3D visualization, Mayavi, version 3.0 or greater, is required.

For a demo, see the 'demo_plot_map' function.

118.2 Functions

`nipy.labs.viz_tools.activation_maps.`**`demo_plot_map`**(*do3d=False*, ***kwargs*)

Demo activation map plotting.

`nipy.labs.viz_tools.activation_maps.`**`plot_anat`**(*anat=None*, *anat_affine=None*, *cut_coords=None*, *slicer='ortho'*, *figure=None*, *axes=None*, *title=None*, *annotate=True*, *draw_cross=True*, *black_bg=False*, *dim=False*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, ***imshow_kwargs*)

Plot three cuts of an anatomical image (Frontal, Axial, and Lateral)

Parameters

anat
[3D ndarray, optional] The anatomical image to be used as a background. If None is given, nipy tries to find a T1 template.

anat_affine
[4x4 ndarray, optional] The affine matrix going from the anatomical image voxel space to MNI space. This parameter is not used when the default anatomical is used, but it is compulsory when using an explicit anatomical image.

figure
[integer or matplotlib figure, optional] Matplotlib figure used or its number. If None is given, a new figure is created.

cut_coords: None, or a tuple of floats
The MNI coordinates of the point where the cut is performed, in MNI coordinates and order. If slicer is 'ortho', this should be a 3-tuple: (x, y, z) For slicer == 'x', 'y', or 'z', then these are the coordinates of each cut in the corresponding direction. If None is given, the cuts is calculated automatically.

slicer: {'ortho', 'x', 'y', 'z'}
Choose the direction of the cuts. With 'ortho' three cuts are performed in orthogonal directions

figure
[integer or matplotlib figure, optional] Matplotlib figure used or its number. If None is given, a new figure is created.

axes
[matplotlib axes or 4 tuple of float: (xmin, ymin, width, height), optional] The axes, or the coordinates, in matplotlib figure space, of the axes used to display the plot. If None, the complete figure is used.

title
[string, optional] The title displayed on the figure.

annotate: boolean, optional
If annotate is True, positions and left/right annotation are added to the plot.

draw_cross: boolean, optional
If draw_cross is True, a cross is drawn on the plot to indicate the cut plosition.

black_bg: boolean, optional
If True, the background of the image is set to be black. If you wish to save figures with a black background, you will need to pass "facecolor='k', edgecolor='k'" to pyplot's savefig.

dim: float, optional
If set, dim the anatomical image, such that vmax = vmean + (1+dim)*ptp if black_bg is set to True, or vmin = vmean - (1+dim)*ptp otherwise, where ptp = .5*(vmax - vmin)

cmap: matplotlib colormap, optional
The colormap for the anat

imshow_kwargs: extra keyword arguments, optional
Extra keyword arguments passed to pyplot.imshow

Notes

Arrays should be passed in numpy convention: (x, y, z) ordered.

`nipy.labs.viz_tools.activation_maps.`**`plot_map`**(*map*, *affine*, *cut_coords=None*, *anat=None*, *anat_affine=None*, *slicer='ortho'*, *figure=None*, *axes=None*, *title=None*, *threshold=None*, *annotate=True*, *draw_cross=True*, *do3d=False*, *threshold_3d=None*, *view_3d=(38.5, 70.5, 300, (-2.7, -12, 9.1))*, *black_bg=False*, ***imshow_kwargs*)

Plot three cuts of a given activation map (Frontal, Axial, and Lateral)

Parameters

map
[3D ndarray] The activation map, as a 3D image.

affine
[4x4 ndarray] The affine matrix going from image voxel space to MNI space.

cut_coords: None, int, or a tuple of floats
The MNI coordinates of the point where the cut is performed, in MNI coordinates and order. If slicer is 'ortho', this should be a 3-tuple: (x, y, z) For slicer == 'x', 'y', or 'z', then these are the coordinates of each cut in the corresponding direction. If None or an int is given,

then a maximally separated sequence (with exactly cut_coords elements if cut_coords is not None) of cut coordinates along the slicer axis is computed automatically

anat
[3D ndarray or False, optional] The anatomical image to be used as a background. If None, the MNI152 T1 1mm template is used. If False, no anat is displayed.

anat_affine
[4x4 ndarray, optional] The affine matrix going from the anatomical image voxel space to MNI space. This parameter is not used when the default anatomical is used, but it is compulsory when using an explicit anatomical image.

slicer: {'ortho', 'x', 'y', 'z'}
Choose the direction of the cuts. With 'ortho' three cuts are performed in orthogonal directions

figure
[integer or matplotlib figure, optional] Matplotlib figure used or its number. If None is given, a new figure is created.

axes
[matplotlib axes or 4 tuple of float: (xmin, ymin, width, height), optional] The axes, or the coordinates, in matplotlib figure space, of the axes used to display the plot. If None, the complete figure is used.

title
[string, optional] The title displayed on the figure.

threshold
[a number, None, or 'auto'] If None is given, the maps are not thresholded. If a number is given, it is used to threshold the maps: values below the threshold are plotted as transparent. If auto is given, the threshold is determined magically by analysis of the map.

annotate: boolean, optional
If annotate is True, positions and left/right annotation are added to the plot.

draw_cross: boolean, optional
If draw_cross is True, a cross is drawn on the plot to indicate the cut plosition.

do3d: {True, False or 'interactive'}, optional
If True, Mayavi is used to plot a 3D view of the map in addition to the slicing. If 'interactive', the 3D visualization is displayed in an additional interactive window.

threshold_3d:
The threshold to use for the 3D view (if any). Defaults to the same threshold as that used for the 2D view.

view_3d: tuple,
The view used to take the screenshot: azimuth, elevation, distance and focalpoint, see the docstring of mlab.view.

black_bg: boolean, optional
If True, the background of the image is set to be black. If you wish to save figures with a black background, you will need to pass "facecolor='k', edgecolor='k'" to pyplot's savefig.

imshow_kwargs: extra keyword arguments, optional
Extra keyword arguments passed to pyplot.imshow

Notes

Arrays should be passed in numpy convention: (x, y, z) ordered.

Use masked arrays to create transparency:

> import numpy as np map = np.ma.masked_less(map, 0.5) plot_map(map, affine)

CHAPTER

NINE

LABS.VIZ_TOOLS.ANAT_CACHE

119.1 Module: `labs.viz_tools.anat_cache`

3D visualization of activation maps using Mayavi

`nipy.labs.viz_tools.anat_cache.`**`find_mni_template`**`()`

Try to find an MNI template on the disk.

CHAPTER

LABS.VIZ_TOOLS.CM

120.1 Module: `labs.viz_tools.cm`

Matplotlib colormaps useful for neuroimaging.

120.2 Functions

`nipy.labs.viz_tools.cm.`**`alpha_cmap`**(*color*, *name=''*)

Return a colormap with the given color, and alpha going from zero to 1.

Parameters

color: (r, g, b), or a string
A triplet of floats ranging from 0 to 1, or a matplotlib color string

`nipy.labs.viz_tools.cm.`**`dim_cmap`**(*cmap*, *factor=0.3*, *to_white=True*)

Dim a colormap to white, or to black.

`nipy.labs.viz_tools.cm.`**`replace_inside`**(*outer_cmap*, *inner_cmap*, *vmin*, *vmax*)

Replace a colormap by another inside a pair of values.

CHAPTER

ONE

LABS.VIZ_TOOLS.COORD_TOOLS

121.1 Module: `labs.viz_tools.coord_tools`

Misc tools to find activations and cut on maps

121.2 Functions

`nipy.labs.viz_tools.coord_tools.`**`coord_transform`**`(`*x*, *y*, *z*, *affine*`)`

Convert x, y, z coordinates from one image space to another space.

Warning: x, y and z have Talairach ordering, not 3D numpy image ordering.

Parameters

x
[number or ndarray] The x coordinates in the input space

y
[number or ndarray] The y coordinates in the input space

z
[number or ndarray] The z coordinates in the input space

affine
[2D 4x4 ndarray] affine that maps from input to output space.

Returns

x
[number or ndarray] The x coordinates in the output space

y
[number or ndarray] The y coordinates in the output space

z
[number or ndarray] The z coordinates in the output space

`nipy.labs.viz_tools.coord_tools.`**`find_cut_coords`**`(`*map*, *mask=None*, *activation_threshold=None*`)`

Find the center of the largest activation connect component.

Parameters

map
[3D ndarray] The activation map, as a 3D image.

mask
[3D ndarray, boolean, optional] An optional brain mask.

activation_threshold
[float, optional] The lower threshold to the positive activation. If None, the activation threshold is computed using find_activation.

Returns

x: float
the x coordinate in voxels.

y: float
the y coordinate in voxels.

z: float
the z coordinate in voxels.

`nipy.labs.viz_tools.coord_tools.`**`find_maxsep_cut_coords`**(*map3d*, *affine*, *slicer='z'*, *n_cuts=None*, *threshold=None*)

Heuristic finds *n_cuts* with max separation along a given axis

Parameters

map3d
[3D array] the data under consideration

affine
[array shape (4, 4)] Affine mapping between array coordinates of *map3d* and real-world coordinates.

slicer
[string, optional] sectional slicer; possible values are "x", "y", or "z"

n_cuts
[None or int >= 1, optional] Number of cuts in the plot; if None, then a default value of 5 is forced.

threshold
[None or float, optional] Thresholding to be applied to the map. Values less than *threshold* set to 0. If None, no thresholding applied.

Returns

cuts
[1D array of length *n_cuts*] the computed cuts

Raises

ValueError:
If *slicer* not in 'xyz'

ValueError
If *ncuts* < 1

`nipy.labs.viz_tools.coord_tools.`**`get_mask_bounds`**(*mask*, *affine*)

Return the world-space bounds occupied by a mask given an affine.

Notes

The mask should have only one connect component.

The affine should be diagonal or diagonal-permuted.

CHAPTER

TWO

LABS.VIZ_TOOLS.MAPS_3D

122.1 Module: `labs.viz_tools.maps_3d`

3D visualization of activation maps using Mayavi

122.2 Functions

`nipy.labs.viz_tools.maps_3d.`**`affine_img_src`**`(`*data*, *affine*, *scale=1*, *name='AffineImage'*, *reverse_x=False*`)`

Make a Mayavi source defined by a 3D array and an affine, for which the voxel of the 3D array are mapped by the affine.

Parameters

data: 3D ndarray
The data arrays

affine: (4 x 4) ndarray
The (4 x 4) affine matrix relating voxels to world coordinates.

scale: float, optional
An optional addition scaling factor.

name: string, optional
The name of the Mayavi source created.

reverse_x: boolean, optional
Reverse the x (lateral) axis. Useful to compared with images in radiologic convention.

Notes

The affine should be diagonal.

`nipy.labs.viz_tools.maps_3d.`**`autocrop_img`**`(`*img*, *bg_color*`)`

`nipy.labs.viz_tools.maps_3d.`**`demo_plot_map_3d`**`()`

`nipy.labs.viz_tools.maps_3d.`**`m2screenshot`**`(`*mayavi_fig=None*, *mpl_axes=None*, *autocrop=True*`)`

Capture a screeshot of the Mayavi figure and display it in the matplotlib axes.

`nipy.labs.viz_tools.maps_3d.`**`plot_anat_3d`**`(`*anat=None*, *anat_affine=None*, *scale=1*, *sulci_opacity=0.5*, *gyri_opacity=0.3*, *opacity=None*, *skull_percentile=78*, *wm_percentile=79*, *outline_color=None*`)`

3D anatomical display

Parameters

skull_percentile
[float, optional] The percentile of the values in the image that delimit the skull from the outside of the brain. The smaller the fraction of you field of view is occupied by the brain, the larger this value should be.

wm_percentile
[float, optional] The percentile of the values in the image that delimit the white matter from the grey matter. Typical this is skull_percentile + 1

nipy.labs.viz_tools.maps_3d.**plot_map_3d**(*map*, *affine*, *cut_coords=None*, *anat=None*, *anat_affine=None*, *threshold=None*, *offscreen=False*, *vmin=None*, *vmax=None*, *cmap=None*, *view=(38.5, 70.5, 300, (-2.7, -12, 9.1))*)

Plot a 3D volume rendering view of the activation, with an outline of the brain.

Parameters

map
[3D ndarray] The activation map, as a 3D image.

affine
[4x4 ndarray] The affine matrix going from image voxel space to MNI space.

cut_coords: 3-tuple of floats, optional
The MNI coordinates of a 3D cursor to indicate a feature or a cut, in MNI coordinates and order.

anat
[3D ndarray, optional] The anatomical image to be used as a background. If None, the MNI152 T1 1mm template is used. If False, no anatomical image is used.

anat_affine
[4x4 ndarray, optional] The affine matrix going from the anatomical image voxel space to MNI space. This parameter is not used when the default anatomical is used, but it is compulsory when using an explicit anatomical image.

threshold
[float, optional] The lower threshold of the positive activation. This parameter is used to threshold the activation map.

offscreen: boolean, optional
If True, Mayavi attempts to plot offscreen. Will work only with VTK >= 5.2.

vmin
[float, optional] The minimal value, for the colormap

vmax
[float, optional] The maximum value, for the colormap

cmap
[a callable, or a pyplot colormap] A callable returning a (n, 4) array for n values between 0 and 1 for the colors. This can be for instance a pyplot colormap.

Notes

If you are using a VTK version below 5.2, there is no way to avoid opening a window during the rendering under Linux. This is necessary to use the graphics card for the rendering. You must maintain this window on top of others and on the screen.

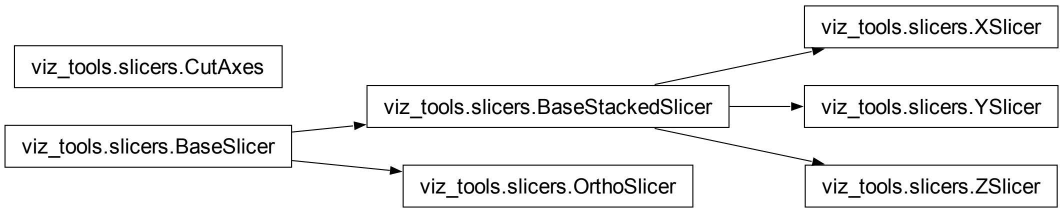
CHAPTER

THREE

LABS.VIZ_TOOLS.SLICERS

123.1 Module: `labs.viz_tools.slicers`

Inheritance diagram for `nipy.labs.viz_tools.slicers`:



viz_tools.slicers.CutAxes
viz_tools.slicers.BaseSlicer
viz_tools.slicers.BaseStackedSlicer
viz_tools.slicers.OrthoSlicer
viz_tools.slicers.XSlicer
viz_tools.slicers.YSlicer
viz_tools.slicers.ZSlicer

The Slicer classes.

The main purpose of these classes is to have auto adjust of axes size to the data with different layout of cuts.

123.2 Classes

123.2.1 `BaseSlicer`

class `nipy.labs.viz_tools.slicers.`**`BaseSlicer`**(*cut_coords*, *axes=None*, *black_bg=False*)

Bases: `object`

The main purpose of these class is to have auto adjust of axes size to the data with different layout of cuts.

`__init__`(*cut_coords*, *axes=None*, *black_bg=False*)

Create 3 linked axes for plotting orthogonal cuts.

Parameters

cut_coords: 3 tuple of ints
The cut position, in world space.

axes: matplotlib axes object, optional
The axes that will be subdivided in 3.

black_bg: boolean, optional
If True, the background of the figure will be put to black. If you wish to save figures with a black background, you will need to pass "facecolor='k', edgecolor='k'" to pyplot's savefig.

`annotate`(*left_right=True*, *positions=True*, *size=12*, ***kwargs*)

Add annotations to the plot.

Parameters

left_right: boolean, optional
If left_right is True, annotations indicating which side is left and which side is right are drawn.

positions: boolean, optional
If positions is True, annotations indicating the positions of the cuts are drawn.

size: integer, optional
The size of the text used.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

`contour_map`(*map*, *affine*, ***kwargs*)

Contour a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

kwargs:
Extra keyword arguments are passed to contour.

`edge_map`(*map*, *affine*, *color='r'*)

Plot the edges of a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

color: matplotlib color: string or (r, g, b) value
The color used to display the edge map

`static find_cut_coords`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*)

`classmethod init_with_figure`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*, *figure=None*, *axes=None*, *black_bg=False*, *leave_space=False*)

`plot_map`(*map*, *affine*, *threshold=None*, ***kwargs*)

Plot a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

threshold
[a number, None, or 'auto'] If None is given, the maps are not thresholded. If a number is given, it is used to threshold the maps: values below the threshold are plotted as transparent.

kwargs:
Extra keyword arguments are passed to imshow.

`title`(*text*, *x=0.01*, *y=0.99*, *size=15*, *color=None*, *bgcolor=None*, *alpha=1*, ***kwargs*)
Write a title to the view.

Parameters

text: string
The text of the title

x: float, optional
The horizontal position of the title on the frame in fraction of the frame width.

y: float, optional
The vertical position of the title on the frame in fraction of the frame height.

size: integer, optional
The size of the title text.

color: matplotlib color specifier, optional
The color of the font of the title.

bgcolor: matplotlib color specifier, optional
The color of the background of the title.

alpha: float, optional
The alpha value for the background.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

123.2.2 BaseStackedSlicer

class `nipy.labs.viz_tools.slicers.BaseStackedSlicer`(*cut_coords*, *axes=None*, *black_bg=False*)
Bases: `BaseSlicer`

A class to create linked axes for plotting stacked cuts of 3D maps.

Notes

The extent of the different axes are adjusted to fit the data best in the viewing area.

Attributes

axes: dictionary of axes
The axes used to plot each view.

frame_axes: axes
The axes framing the whole set of views.

`__init__`(*cut_coords*, *axes=None*, *black_bg=False*)

Create 3 linked axes for plotting orthogonal cuts.

Parameters

cut_coords: 3 tuple of ints
The cut position, in world space.

axes: matplotlib axes object, optional
The axes that will be subdivided in 3.

black_bg: boolean, optional
If True, the background of the figure will be put to black. If you wish to save figures with a black background, you will need to pass "facecolor='k', edgecolor='k'" to pyplot's savefig.

`annotate`(*left_right=True*, *positions=True*, *size=12*, ***kwargs*)

Add annotations to the plot.

Parameters

left_right: boolean, optional
If left_right is True, annotations indicating which side is left and which side is right are drawn.

positions: boolean, optional
If positions is True, annotations indicating the positions of the cuts are drawn.

size: integer, optional
The size of the text used.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

`contour_map`(*map*, *affine*, ***kwargs*)

Contour a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

kwargs:
Extra keyword arguments are passed to contour.

`draw_cross`(*cut_coords=None*, ***kwargs*)

Draw a crossbar on the plot to show where the cut is performed.

Parameters

cut_coords: 3-tuple of floats, optional
The position of the cross to draw. If none is passed, the ortho_slicer's cut coordinates are used.

kwargs:
Extra keyword arguments are passed to axhline

`edge_map`(*map*, *affine*, *color='r'*)

Plot the edges of a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

color: matplotlib color: string or (r, g, b) value
The color used to display the edge map

classmethod `find_cut_coords`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*)

classmethod `init_with_figure`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*, *figure=None*, *axes=None*, *black_bg=False*, *leave_space=False*)

`plot_map`(*map*, *affine*, *threshold=None*, ***kwargs*)

Plot a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

threshold
[a number, None, or 'auto'] If None is given, the maps are not thresholded. If a number is given, it is used to threshold the maps: values below the threshold are plotted as transparent.

kwargs:
Extra keyword arguments are passed to imshow.

`title`(*text*, *x=0.01*, *y=0.99*, *size=15*, *color=None*, *bgcolor=None*, *alpha=1*, ***kwargs*)

Write a title to the view.

Parameters

text: string
The text of the title

x: float, optional
The horizontal position of the title on the frame in fraction of the frame width.

y: float, optional
The vertical position of the title on the frame in fraction of the frame height.

size: integer, optional
The size of the title text.

color: matplotlib color specifier, optional
The color of the font of the title.

bgcolor: matplotlib color specifier, optional
The color of the background of the title.

alpha: float, optional
The alpha value for the background.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

123.2.3 CutAxes

class nipy.labs.viz_tools.slicers.**CutAxes**(*ax*, *direction*, *coord*)

Bases: `object`

An MPL axis-like object that displays a cut of 3D volumes

`__init__`(*ax*, *direction*, *coord*)

An MPL axis-like object that displays a cut of 3D volumes

Parameters

ax: a MPL axes instance
The axes in which the plots will be drawn

direction: {'x', 'y', 'z'}
The directions of the cut

coord: float
The coordinate along the direction of the cut

`do_cut`(*map*, *affine*)

Cut the 3D volume into a 2D slice

Parameters

map: 3D ndarray
The 3D volume to cut

affine: 4x4 ndarray
The affine of the volume

`draw_cut`(*cut*, *data_bounds*, *bounding_box*, *type='imshow'*, ***kwargs*)

`draw_left_right`(*size*, *bg_color*, ***kwargs*)

`draw_position`(*size*, *bg_color*, ***kwargs*)

`get_object_bounds`()

Return the bounds of the objects on this axes.

123.2.4 OrthoSlicer

class nipy.labs.viz_tools.slicers.**OrthoSlicer**(*cut_coords*, *axes=None*, *black_bg=False*)

Bases: *BaseSlicer*

A class to create 3 linked axes for plotting orthogonal cuts of 3D maps.

Notes

The extent of the different axes are adjusted to fit the data best in the viewing area.

Attributes

axes: dictionary of axes
The 3 axes used to plot each view.

frame_axes: axes
The axes framing the whole set of views.

`__init__`(*cut_coords*, *axes=None*, *black_bg=False*)

Create 3 linked axes for plotting orthogonal cuts.

Parameters

cut_coords: 3 tuple of ints
The cut position, in world space.

axes: matplotlib axes object, optional
The axes that will be subdivided in 3.

black_bg: boolean, optional
If True, the background of the figure will be put to black. If you wish to save figures with a black background, you will need to pass "facecolor='k', edgecolor='k'" to pyplot's savefig.

`annotate`(*left_right=True*, *positions=True*, *size=12*, ***kwargs*)

Add annotations to the plot.

Parameters

left_right: boolean, optional
If left_right is True, annotations indicating which side is left and which side is right are drawn.

positions: boolean, optional
If positions is True, annotations indicating the positions of the cuts are drawn.

size: integer, optional
The size of the text used.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

`contour_map`(*map*, *affine*, ***kwargs*)

Contour a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

kwargs:
Extra keyword arguments are passed to contour.

`draw_cross`(*cut_coords=None*, ***kwargs*)

Draw a crossbar on the plot to show where the cut is performed.

Parameters

cut_coords: 3-tuple of floats, optional
The position of the cross to draw. If none is passed, the ortho_slicer's cut coordinates are used.

kwargs:
Extra keyword arguments are passed to axhline

`edge_map`(*map*, *affine*, *color='r'*)

Plot the edges of a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

color: matplotlib color: string or (r, g, b) value
The color used to display the edge map

`static find_cut_coords`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*)

`classmethod init_with_figure`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*, *figure=None*, *axes=None*, *black_bg=False*, *leave_space=False*)

`plot_map`(*map*, *affine*, *threshold=None*, ***kwargs*)
Plot a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

threshold
[a number, None, or 'auto'] If None is given, the maps are not thresholded. If a number is given, it is used to threshold the maps: values below the threshold are plotted as transparent.

kwargs:
Extra keyword arguments are passed to imshow.

`title`(*text*, *x=0.01*, *y=0.99*, *size=15*, *color=None*, *bgcolor=None*, *alpha=1*, ***kwargs*)
Write a title to the view.

Parameters

text: string
The text of the title

x: float, optional
The horizontal position of the title on the frame in fraction of the frame width.

y: float, optional
The vertical position of the title on the frame in fraction of the frame height.

size: integer, optional
The size of the title text.

color: matplotlib color specifier, optional
The color of the font of the title.

bgcolor: matplotlib color specifier, optional
The color of the background of the title.

alpha: float, optional
The alpha value for the background.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

123.2.5 `XSlicer`

class `nipy.labs.viz_tools.slicers.`**`XSlicer`**(*cut_coords*, *axes=None*, *black_bg=False*)

Bases: *BaseStackedSlicer*

`__init__`(*cut_coords*, *axes=None*, *black_bg=False*)

Create 3 linked axes for plotting orthogonal cuts.

Parameters

cut_coords: 3 tuple of ints
The cut position, in world space.

axes: matplotlib axes object, optional
The axes that will be subdivided in 3.

black_bg: boolean, optional
If True, the background of the figure will be put to black. If you wish to save figures with a black background, you will need to pass "facecolor='k', edgecolor='k'" to pyplot's savefig.

`annotate`(*left_right=True*, *positions=True*, *size=12*, ***kwargs*)

Add annotations to the plot.

Parameters

left_right: boolean, optional
If left_right is True, annotations indicating which side is left and which side is right are drawn.

positions: boolean, optional
If positions is True, annotations indicating the positions of the cuts are drawn.

size: integer, optional
The size of the text used.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

`contour_map`(*map*, *affine*, ***kwargs*)

Contour a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

kwargs:
Extra keyword arguments are passed to contour.

`draw_cross`(*cut_coords=None*, ***kwargs*)

Draw a crossbar on the plot to show where the cut is performed.

Parameters

cut_coords: 3-tuple of floats, optional
The position of the cross to draw. If none is passed, the ortho_slicer's cut coordinates are used.

kwargs:
Extra keyword arguments are passed to axhline

`edge_map`(*map*, *affine*, *color='r'*)

Plot the edges of a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

color: matplotlib color: string or (r, g, b) value
The color used to display the edge map

`classmethod find_cut_coords`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*)

`classmethod init_with_figure`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*, *figure=None*, *axes=None*, *black_bg=False*, *leave_space=False*)

`plot_map`(*map*, *affine*, *threshold=None*, ***kwargs*)

Plot a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

threshold
[a number, None, or 'auto'] If None is given, the maps are not thresholded. If a number is given, it is used to threshold the maps: values below the threshold are plotted as transparent.

kwargs:
Extra keyword arguments are passed to imshow.

`title`(*text*, *x=0.01*, *y=0.99*, *size=15*, *color=None*, *bgcolor=None*, *alpha=1*, ***kwargs*)

Write a title to the view.

Parameters

text: string
The text of the title

x: float, optional
The horizontal position of the title on the frame in fraction of the frame width.

y: float, optional
The vertical position of the title on the frame in fraction of the frame height.

size: integer, optional
The size of the title text.

color: matplotlib color specifier, optional
The color of the font of the title.

bgcolor: matplotlib color specifier, optional
The color of the background of the title.

alpha: float, optional
The alpha value for the background.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

123.2.6 YSlicer

class nipy.labs.viz_tools.slicers.**YSlicer**(*cut_coords*, *axes=None*, *black_bg=False*)

Bases: *BaseStackedSlicer*

__init__(*cut_coords*, *axes=None*, *black_bg=False*)

Create 3 linked axes for plotting orthogonal cuts.

Parameters

cut_coords: 3 tuple of ints
The cut position, in world space.

axes: matplotlib axes object, optional
The axes that will be subdivided in 3.

black_bg: boolean, optional
If True, the background of the figure will be put to black. If you wish to save figures with a black background, you will need to pass "facecolor='k', edgecolor='k'" to pyplot's savefig.

annotate(*left_right=True*, *positions=True*, *size=12*, ***kwargs*)

Add annotations to the plot.

Parameters

left_right: boolean, optional
If left_right is True, annotations indicating which side is left and which side is right are drawn.

positions: boolean, optional
If positions is True, annotations indicating the positions of the cuts are drawn.

size: integer, optional
The size of the text used.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

contour_map(*map*, *affine*, ***kwargs*)

Contour a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

kwargs:
Extra keyword arguments are passed to contour.

draw_cross(*cut_coords=None*, ***kwargs*)

Draw a crossbar on the plot to show where the cut is performed.

Parameters

cut_coords: 3-tuple of floats, optional
The position of the cross to draw. If none is passed, the ortho_slicer's cut coordinates are used.

kwargs:
Extra keyword arguments are passed to axhline

`edge_map`(*map*, *affine*, *color='r'*)

Plot the edges of a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

color: matplotlib color: string or (r, g, b) value
The color used to display the edge map

`classmethod find_cut_coords`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*)

`classmethod init_with_figure`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*, *figure=None*, *axes=None*, *black_bg=False*, *leave_space=False*)

`plot_map`(*map*, *affine*, *threshold=None*, ***kwargs*)

Plot a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

threshold
[a number, None, or 'auto'] If None is given, the maps are not thresholded. If a number is given, it is used to threshold the maps: values below the threshold are plotted as transparent.

kwargs:
Extra keyword arguments are passed to imshow.

`title`(*text*, *x=0.01*, *y=0.99*, *size=15*, *color=None*, *bgcolor=None*, *alpha=1*, ***kwargs*)

Write a title to the view.

Parameters

text: string
The text of the title

x: float, optional
The horizontal position of the title on the frame in fraction of the frame width.

y: float, optional
The vertical position of the title on the frame in fraction of the frame height.

size: integer, optional
The size of the title text.

color: matplotlib color specifier, optional
The color of the font of the title.

bgcolor: matplotlib color specifier, optional
The color of the background of the title.

alpha: float, optional
The alpha value for the background.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

123.2.7 ZSlicer

class nipy.labs.viz_tools.slicers.**ZSlicer**(*cut_coords*, *axes=None*, *black_bg=False*)

Bases: *BaseStackedSlicer*

__init__(*cut_coords*, *axes=None*, *black_bg=False*)

Create 3 linked axes for plotting orthogonal cuts.

Parameters

cut_coords: 3 tuple of ints
The cut position, in world space.

axes: matplotlib axes object, optional
The axes that will be subdivided in 3.

black_bg: boolean, optional
If True, the background of the figure will be put to black. If you wish to save figures with a black background, you will need to pass "facecolor='k', edgecolor='k'" to pyplot's savefig.

annotate(*left_right=True*, *positions=True*, *size=12*, ***kwargs*)

Add annotations to the plot.

Parameters

left_right: boolean, optional
If left_right is True, annotations indicating which side is left and which side is right are drawn.

positions: boolean, optional
If positions is True, annotations indicating the positions of the cuts are drawn.

size: integer, optional
The size of the text used.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

contour_map(*map*, *affine*, ***kwargs*)

Contour a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

kwargs:
Extra keyword arguments are passed to contour.

`draw_cross`(*cut_coords=None*, ***kwargs*)

Draw a crossbar on the plot to show where the cut is performed.

Parameters

cut_coords: 3-tuple of floats, optional
The position of the cross to draw. If none is passed, the ortho_slicer's cut coordinates are used.

kwargs:
Extra keyword arguments are passed to axhline

`edge_map`(*map*, *affine*, *color='r'*)

Plot the edges of a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

color: matplotlib color: string or (r, g, b) value
The color used to display the edge map

`classmethod find_cut_coords`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*)

`classmethod init_with_figure`(*data=None*, *affine=None*, *threshold=None*, *cut_coords=None*, *figure=None*, *axes=None*, *black_bg=False*, *leave_space=False*)

`plot_map`(*map*, *affine*, *threshold=None*, ***kwargs*)

Plot a 3D map in all the views.

Parameters

map: 3D ndarray
The 3D map to be plotted. If it is a masked array, only the non-masked part will be plotted.

affine: 4x4 ndarray
The affine matrix giving the transformation from voxel indices to world space.

threshold
[a number, None, or 'auto'] If None is given, the maps are not thresholded. If a number is given, it is used to threshold the maps: values below the threshold are plotted as transparent.

kwargs:
Extra keyword arguments are passed to imshow.

`title`(*text*, *x=0.01*, *y=0.99*, *size=15*, *color=None*, *bgcolor=None*, *alpha=1*, ***kwargs*)

Write a title to the view.

Parameters

text: string
The text of the title

x: float, optional
The horizontal position of the title on the frame in fraction of the frame width.

y: float, optional
The vertical position of the title on the frame in fraction of the frame height.

size: integer, optional
The size of the title text.

color: matplotlib color specifier, optional
The color of the font of the title.

bgcolor: matplotlib color specifier, optional
The color of the background of the title.

alpha: float, optional
The alpha value for the background.

kwargs:
Extra keyword arguments are passed to matplotlib's text function.

123.3 Function

`nipy.labs.viz_tools.slicers.demo_ortho_slicer()`

A small demo of the OrthoSlicer functionality.

CHAPTER

FOUR

LABS.VIZ_TOOLS.TEST.TEST_ACTIVATION_MAPS

124.1 Module: `labs.viz_tools.test.test_activation_maps`

124.2 Functions

nipy.labs.viz_tools.test.test_activation_maps.**test_anat_cache**()

nipy.labs.viz_tools.test.test_activation_maps.**test_demo_plot_map**()

nipy.labs.viz_tools.test.test_activation_maps.**test_plot_anat**()

nipy.labs.viz_tools.test.test_activation_maps.**test_plot_anat_kwargs**()

nipy.labs.viz_tools.test.test_activation_maps.**test_plot_map_empty**()

nipy.labs.viz_tools.test.test_activation_maps.**test_plot_map_with_auto_cut_coords**()

CHAPTER

FIVE

LABS.VIZ_TOOLS.TEST.TEST_CM

125.1 Module: `labs.viz_tools.test.test_cm`

Smoke testing the cm module

125.2 Functions

`nipy.labs.viz_tools.test.test_cm.`**`test_dim_cmap`**`()`

`nipy.labs.viz_tools.test.test_cm.`**`test_replace_inside`**`()`

CHAPTER

SIX

LABS.VIZ_TOOLS.TEST.TEST_COORD_TOOLS

126.1 Module: `labs.viz_tools.test.test_coord_tools`

126.2 Functions

nipy.labs.viz_tools.test.test_coord_tools.**test_coord_transform_trivial**()

nipy.labs.viz_tools.test.test_coord_tools.**test_find_cut_coords**()

nipy.labs.viz_tools.test.test_coord_tools.**test_find_maxsep_cut_coords**()

CHAPTER

SEVEN

LABS.VIZ_TOOLS.TEST.TEST_EDGE_DETECT

127.1 Module: `labs.viz_tools.test.test_edge_detect`

127.2 Functions

`nipy.labs.viz_tools.test.test_edge_detect.`**`test_edge_detect`**`()`

`nipy.labs.viz_tools.test.test_edge_detect.`**`test_fast_abs_percentile`**`()`

CHAPTER

EIGHT

LABS.VIZ_TOOLS.TEST.TEST_SLICERS

128.1 Module: `labs.viz_tools.test.test_slicers`

nipy.labs.viz_tools.test.test_slicers.**test_demo_ortho_slicer**()

CHAPTER

NINE

MODALITIES.FMRI.DESIGN

129.1 Module: `modalities.fmri.design`

Convenience functions for specifying a design in the GLM

129.2 Functions

`nipy.modalities.fmri.design.`**`block_amplitudes`**(*name*, *block_spec*, *t*, *hrfs=(glover,)*, *convolution_padding=5.0*, *convolution_dt=0.02*, *hrf_interval=(0.0, 30.0)*)

Design matrix at times *t* for blocks specification *block_spec*

Create design matrix for linear model from a block specification *block_spec*, evaluating design rows at a sequence of time values *t*.

block_spec may specify amplitude of response for each event, if different (see description of *block_spec* parameter below).

The on-off step function implied by *block_spec* will be convolved with each HRF in *hrfs* to form a design matrix shape `(len(t), len(hrfs))`.

Parameters

name
[str] Name of condition

block_spec
[np.recarray or array-like] A recarray having fields `start, end, amplitude`, or a 2D ndarray / array-like with three columns corresponding to start, end, amplitude.

t
[np.ndarray] An array of np.float64 values at which to evaluate the design. Common examples would be the acquisition times of an fMRI image.

hrfs
[sequence, optional] A sequence of (symbolic) HRFs that will be convolved with each block. Default is `(glover,)`.

convolution_padding
[float, optional] A padding for the convolution with the HRF. The intervals used for the convolution are the smallest 'start' minus this padding to the largest 'end' plus this padding.

convolution_dt
[float, optional] Time step for high-resolution time course for use in convolving the blocks with each HRF.

hrf_interval: length 2 sequence of floats, optional
Interval over which the HRF is assumed supported, used in the convolution.

Returns

X
[np.ndarray] The design matrix with `X.shape[0] == t.shape[0]`. The number of columns will be `len(hrfs)`.

contrasts
[dict] A contrast is generated for each HRF specified in *hrfs*.

`nipy.modalities.fmri.design.`**`block_design`**(*block_spec*, *t*, *order=2*, *hrfs=(glover,)*, *convolution_padding=5.0*, *convolution_dt=0.02*, *hrf_interval=(0.0, 30.0)*, *level_contrasts=False*)

Create design matrix at times *t* for blocks specification *block_spec*

Create design matrix for linear model from a block specification *block_spec*, evaluating design rows at a sequence of time values *t*. Each column in the design matrix will be convolved with each HRF in *hrfs*.

Parameters

block_spec
[np.recarray] A recarray having at least a field named 'start' and a field named 'end' signifying the block onset and offset times. All other fields will be treated as factors in an ANOVA-type model. If there is no field other than 'start' and 'end', add a single-level placeholder block type `_block_`.

t
[np.ndarray] An array of np.float64 values at which to evaluate the design. Common examples would be the acquisition times of an fMRI image.

order
[int, optional] The highest order interaction to be considered in constructing the contrast matrices.

hrfs
[sequence, optional] A sequence of (symbolic) HRFs that will be convolved with each block. Default is `(glover,)`.

convolution_padding
[float, optional] A padding for the convolution with the HRF. The intervals used for the convolution are the smallest 'start' minus this padding to the largest 'end' plus this padding.

convolution_dt
[float, optional] Time step for high-resolution time course for use in convolving the blocks with each HRF.

hrf_interval: length 2 sequence of floats, optional
Interval over which the HRF is assumed supported, used in the convolution.

level_contrasts
[bool, optional] If true, generate contrasts for each individual level of each factor.

Returns

X

[np.ndarray] The design matrix with `X.shape[0] == t.shape[0]`. The number of columns will depend on the other fields of *block_spec*.

contrasts

[dict] Dictionary of contrasts that are expected to be of interest from the block specification. Each interaction / effect up to a given order will be returned. Also, a contrast is generated for each interaction / effect for each HRF specified in *hrfs*.

`nipy.modalities.fmri.design.`**`event_design`**(*event_spec*, *t*, *order=2*, *hrfs=(glover,)*, *level_contrasts=False*)

Create design matrix at times *t* for event specification *event_spec*

Create a design matrix for linear model based on an event specification *event_spec*, evaluating the design rows at a sequence of time values *t*. Each column in the design matrix will be convolved with each HRF in *hrfs*.

Parameters

event_spec

[np.recarray] A recarray having at least a field named 'time' signifying the event time, and all other fields will be treated as factors in an ANOVA-type model. If there is no field other than time, add a single-level placeholder event type `_event_`.

t

[np.ndarray] An array of np.float64 values at which to evaluate the design. Common examples would be the acquisition times of an fMRI image.

order

[int, optional] The highest order interaction to be considered in constructing the contrast matrices.

hrfs

[sequence, optional] A sequence of (symbolic) HRFs that will be convolved with each event. Default is `(glover,)`.

level_contrasts

[bool, optional] If True, generate contrasts for each individual level of each factor.

Returns

X

[np.ndarray] The design matrix with `X.shape[0] == t.shape[0]`. The number of columns will depend on the other fields of *event_spec*.

contrasts

[dict] Dictionary of contrasts that is expected to be of interest from the event specification. Each interaction / effect up to a given order will be returned. Also, a contrast is generated for each interaction / effect for each HRF specified in *hrfs*.

`nipy.modalities.fmri.design.`**`fourier_basis`**(*t*, *freq*)

Create a design matrix with columns given by the Fourier basis with a given set of frequencies.

Parameters

t

[np.ndarray] An array of np.float64 values at which to evaluate the design. Common examples would be the acquisition times of an fMRI image.

freq

[sequence of float] Frequencies for the terms in the Fourier basis.

Returns

X
[np.ndarray]

Examples

```
>>> t = np.linspace(0,50,101)
>>> drift = fourier_basis(t, np.array([4,6,8]))
>>> drift.shape
(101, 6)
```

`nipy.modalities.fmri.design.`**`natural_spline`**(*tvals*, *knots=None*, *order=3*, *intercept=True*)

Design matrix with columns given by a natural spline order *order*

Return design matrix with natural splines with knots *knots*, order *order*. If *intercept* == True (the default), add constant column.

Parameters

tvals
[np.array] Time values

knots
[None or sequence, optional] Sequence of float. Default None (same as empty list)

order
[int, optional] Order of the spline. Defaults to a cubic (==3)

intercept
[bool, optional] If True, include a constant function in the natural spline. Default is False

Returns

X
[np.ndarray]

Examples

```
>>> tvals = np.linspace(0,50,101)
>>> drift = natural_spline(tvals, knots=[10,20,30,40])
>>> drift.shape
(101, 8)
```

`nipy.modalities.fmri.design.`**`openfmri2nipy`**(*ons_dur_amp*)

Contents of OpenFMRI condition file *ons_dur_map* as nipy recarray

Parameters

ons_dur_amp
[str or array] Path to OpenFMRI stimulus file or 2D array containing three columns corresponding to onset, duration, amplitude.

Returns

block_spec
[array] Structured array with fields “start” (corresponding to onset time), “end” (onset time plus duration), “amplitude”.

`nipy.modalities.fmri.design.`**`stack2designs`**(*old_X*, *new_X*, *old_contrasts={}*, *new_contrasts={}*)

Add some columns to a design matrix that has contrasts matrices already specified, adding some possibly new contrasts as well.

This basically performs an np.hstack of old_X, new_X and makes sure the contrast matrices are dealt with accordingly.

If two contrasts have the same name, an exception is raised.

Parameters

old_X
[np.ndarray] A design matrix

new_X
[np.ndarray] A second design matrix to be stacked with old_X

old_contrast
[dict] Dictionary of contrasts in the old_X column space

new_contrasts
[dict] Dictionary of contrasts in the new_X column space

Returns

X
[np.ndarray] A new design matrix: np.hstack([old_X, new_X])

contrasts
[dict] The new contrast matrices reflecting changes to the columns.

`nipy.modalities.fmri.design.`**`stack_contrasts`**(*contrasts*, *name*, *keys*)

Create a new F-contrast matrix called 'name' based on a sequence of keys. The contrast is added to contrasts, in-place.

Parameters

contrasts
[dict] Dictionary of contrast matrices

name
[str] Name of new contrast. Should not already be a key of contrasts.

keys
[sequence of str] Keys of contrasts that are to be stacked.

Returns

None

`nipy.modalities.fmri.design.`**`stack_designs`**(**pairs*)

Stack a sequence of design / contrast dictionary pairs

Uses multiple calls to `stack2designs()`

Parameters

***pairs**
[sequence] Elements of either (np.ndarray, dict) or (np.ndarray,) or np.ndarray

Returns

X
[np.ndarray] new design matrix: np.hstack([old_X, new_X])

contrasts
[dict] The new contrast matrices reflecting changes to the columns.

MODALITIES.FMRI.DESIGN_MATRIX

130.1 Module: `modalities.fmri.design_matrix`

Inheritance diagram for `nipy.modalities.fmri.design_matrix`:

This module implements fMRI Design Matrix creation.

The DesignMatrix object is just a container that represents the design matrix. Computations of the different parts of the design matrix are confined to the make_dmtx() function, that instantiates the DesignMatrix object. All the remainder are just ancillary functions.

Design matrices contain three different types of regressors:

1. Task-related regressors, that result from the convolution of the experimental paradigm regressors with hemodynamic models
2. User-specified regressors, that represent information available on the data, e.g. motion parameters, physiological data resampled at the acquisition rate, or sinusoidal regressors that model the signal at a frequency of interest.
3. Drift regressors, that represent low_frequency phenomena of no interest in the data; they need to be included to reduce variance estimates.

Author: Bertrand Thirion, 2009-2011

130.2 Class

130.3 `DesignMatrix`

`class nipy.modalities.fmri.design_matrix.`**`DesignMatrix`**(*matrix*, *names*, *frametimes=None*)

Bases: `object`

This is a container for a light-weight class for design matrices

This class is only used to make IO and visualization.

Attributes

matrix: array of shape (n_scans, n_regressors)
the numerical specification of the matrix.

names: list of len (n_regressors)
the names associated with the columns.

frametimes: array of shape (n_scans), optional
the occurrence time of the matrix rows.

`__init__`(*matrix*, *names*, *frametimes=None*)

`show`(*rescale=True*, *ax=None*, *cmap=None*)

Visualization of a design matrix

Parameters

rescale: bool, optional
rescale columns magnitude for visualization or not.

ax: axis handle, optional
Handle to axis onto which we will draw design matrix.

cmap: colormap, optional
Matplotlib colormap to use, passed to *imshow*.

Returns

ax: axis handle

`show_contrast`(*contrast*, *ax=None*, *cmap=None*)

Plot a contrast for a design matrix.

Parameters

contrast
[np.float64] Array forming contrast with respect to the design matrix.

ax: axis handle, optional
Handle to axis onto which we will draw design matrix.

cmap: colormap, optional
Matplotlib colormap to use, passed to *imshow*.

Returns

ax: axis handle

`write_csv`(*path*)

write self.matrix as a csv file with appropriate column names

Parameters

path: string, path of the resulting csv file

Notes

The frametimes are not written

130.4 Functions

`nipy.modalities.fmri.design_matrix.dmtx_from_csv`(*path*, *frametimes=None*)

Return a DesignMatrix instance from a csv file

Parameters

path: string, path of the .csv file

Returns

A DesignMatrix instance

`nipy.modalities.fmri.design_matrix.dmtx_light`(*frametimes*, *paradigm=None*, *hrf_model='canonical'*, *drift_model='cosine'*, *hfcut=128*, *drift_order=1*, *fir_delays=[0]*, *add_regs=None*, *add_reg_names=None*, *min_onset=-24*, *path=None*)

Make a design matrix while avoiding framework

Parameters

see make_dmtx, plus
path: string, optional: a path to write the output

Returns

dmtx array of shape(nreg, nbframes):
the sampled design matrix

names list of strings of len (nreg)
the names of the columns of the design matrix

`nipy.modalities.fmri.design_matrix.make_dmtx`(*frametimes*, *paradigm=None*, *hrf_model='canonical'*, *drift_model='cosine'*, *hfcut=128*, *drift_order=1*, *fir_delays=[0]*, *add_regs=None*, *add_reg_names=None*, *min_onset=-24*)

Generate a design matrix from the input parameters

Parameters

frametimes: array of shape(nbframes), the timing of the scans
paradigm: Paradigm instance, optional
description of the experimental paradigm

hrf_model: string, optional,
that specifies the hemodynamic response function. Can be one of {'canonical', 'canonical with derivative', 'fir', 'spm', 'spm_time', 'spm_time_dispersion'}.

drift_model: string, optional
specifies the desired drift model, to be chosen among 'polynomial', 'cosine', 'blank'

hfcut: float, optional
cut period of the low-pass filter

drift_order: int, optional
order of the drift model (in case it is polynomial)

fir_delays: array of shape(nb_onsets) or list, optional,
in case of FIR design, yields the array of delays used in the FIR model

add_regs: array of shape(nbframes, naddreg), optional
additional user-supplied regressors

add_reg_names: list of (naddreg) regressor names, optional
if None, while naddreg>0, these will be termed 'reg_%i',i=0..naddreg-1

min_onset: float, optional
minimal onset relative to frametimes[0] (in seconds) events that start before frametimes[0] + min_onset are not considered

Returns

DesignMatrix instance

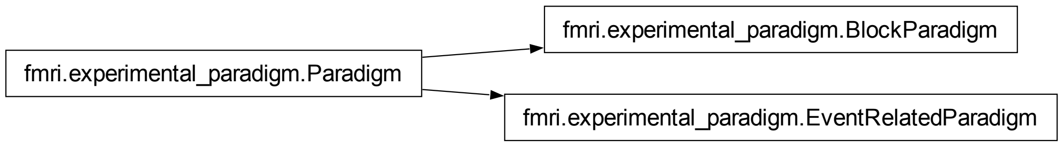
CHAPTER

ONE

MODALITIES.FMRI.EXPERIMENTAL_PARADIGM

131.1 Module: `modalities.fmri.experimental_paradigm`

Inheritance diagram for `nipy.modalities.fmri.experimental_paradigm`:


fmri.experimental_paradigm.Paradigm
fmri.experimental_paradigm.BlockParadigm
fmri.experimental_paradigm.EventRelatedParadigm

This module implements an object to deal with experimental paradigms. In fMRI data analysis, there are two main types of experimental paradigms: block and event-related paradigms. They correspond to 2 classes EventRelatedParadigm and BlockParadigm. Both are implemented here, together with functions to write paradigms to csv files.

131.1.1 Notes

Although the Paradigm object have no notion of session or acquisitions (they are assumed to correspond to a sequential acquisition, called 'session' in SPM jargon), the .csv file used to represent paradigm may be multi-session, so it is assumed that the first column of a file yielding a paradigm is in fact a session index

Author: Bertrand Thirion, 2009-2011

131.2 Classes

131.2.1 `BlockParadigm`

class `nipy.modalities.fmri.experimental_paradigm.`**`BlockParadigm`**(*con_id=None*, *onset=None*, *duration=None*, *amplitude=None*)

Bases: *`Paradigm`*

Class to handle block paradigms

`__init__`(*con_id=None*, *onset=None*, *duration=None*, *amplitude=None*)

Parameters

con_id: array of shape (n_events), type = string, optional
id of the events (name of the experimental condition)

onset: array of shape (n_events), type = float, optional
onset time (in s.) of the events

amplitude: array of shape (n_events), type = float, optional,
amplitude of the events (if applicable)

`write_to_csv`(*csv_file*, *session='0'*)

Write the paradigm to a csv file

Parameters

csv_file: string, path of the csv file
session: string, optional, session identifier

131.2.2 EventRelatedParadigm

class nipy.modalities.fmri.experimental_paradigm.**EventRelatedParadigm**(*con_id=None*, *onset=None*, *amplitude=None*)

Bases: *Paradigm*

Class to handle event-related paradigms

`__init__`(*con_id=None*, *onset=None*, *amplitude=None*)

Parameters

con_id: array of shape (n_events), type = string, optional
id of the events (name of the experimental condition)

onset: array of shape (n_events), type = float, optional
onset time (in s.) of the events

amplitude: array of shape (n_events), type = float, optional,
amplitude of the events (if applicable)

`write_to_csv`(*csv_file*, *session='0'*)

Write the paradigm to a csv file

Parameters

csv_file: string, path of the csv file
session: string, optional, session identifier

131.2.3 `Paradigm`

class `nipy.modalities.fmri.experimental_paradigm.`**`Paradigm`**(*con_id=None*, *onset=None*, *amplitude=None*)

Bases: `object`

Simple class to handle the experimental paradigm in one session

`__init__`(*con_id=None*, *onset=None*, *amplitude=None*)

Parameters

con_id: array of shape (n_events), type = string, optional
identifier of the events

onset: array of shape (n_events), type = float, optional,
onset time (in s.) of the events

amplitude: array of shape (n_events), type = float, optional,
amplitude of the events (if applicable)

`write_to_csv`(*csv_file*, *session='0'*)

Write the paradigm to a csv file

Parameters

csv_file: string, path of the csv file
session: string, optional, session identifier

131.3 Function

`nipy.modalities.fmri.experimental_paradigm.`**`load_paradigm_from_csv_file`**(*path*, *session=None*)

Read a (.csv) paradigm file consisting of values yielding (occurrence time, (duration), event ID, modulation) and returns a paradigm instance or a dictionary of paradigm instances

Parameters

path: string,
path to a .csv file that describes the paradigm

session: string, optional, session identifier
by default the output is a dictionary of session-level dictionaries indexed by session

Returns

paradigm, paradigm instance (if session is provided), or
dictionary of paradigm instances otherwise, the resulting session-by-session paradigm

Notes

It is assumed that the csv file contains the following columns: (session id, condition id, onset), plus possibly (duration) and/or (amplitude). If all the durations are 0, the paradigm will be handled as event-related.

FIXME: would be much clearer if amplitude was put before duration in the .csv

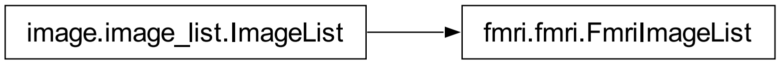
CHAPTER

TWO

MODALITIES.FMRI.FMRI

132.1 Module: `modalities.fmri.fmri`

Inheritance diagram for `nipy.modalities.fmri.fmri`:



132.2 `FmriImageList`

class `nipy.modalities.fmri.fmri.`**`FmriImageList`**(*images=None*, *volume_start_times=None*, *slice_times=None*)

Bases: *`ImageList`*

Class to implement image list interface for FMRI time series

Allows metadata such as volume and slice times

`__init__`(*images=None*, *volume_start_times=None*, *slice_times=None*)

An implementation of an fMRI image as in ImageList

Parameters

images
[iterable] an iterable object whose items are meant to be images; this is checked by asserting that each has a *coordmap* attribute and a `get_fdata` method. Note that Image objects are not iterable by default; use the `from_image` classmethod or `iter_axis` function to convert images to image lists - see examples below for the latter.

volume_start_times: None or float or (N,) ndarray
start time of each frame. It can be specified either as an ndarray with `N=len(images)` elements or as a single float, the TR. None results in `np.arange(len(images)).astype(np.float64)`

slice_times: None or (N,) ndarray
specifying offset for each slice of each frame, from the frame start time.

See also:

`nipy.core.image_list.ImageList`

Examples

```
>>> from nipy.testing import funcfile
>>> from nipy.io.api import load_image
>>> from nipy.core.api import iter_axis
>>> funcim = load_image(funcfile)
>>> iterable_img = iter_axis(funcim, 't')
>>> fmrilist = FmriImageList(iterable_img)
>>> print(fmrilist.get_list_data(axis=0).shape)
(20, 17, 21, 3)
>>> print(fmrilist[4].shape)
(17, 21, 3)
```

classmethod from_image(*fourdimage*, *axis='t'*, *volume_start_times=None*, *slice_times=None*)

Create an FmriImageList from a 4D Image

Get images by extracting 3d images along the 't' axis.

Parameters

fourdimage
[`Image` instance] A 4D Image

volume_start_times: None or float or (N,) ndarray
start time of each frame. It can be specified either as an ndarray with `N=len(images)` elements or as a single float, the TR. None results in `np.arange(len(images)).astype(np.float64)`

slice_times: None or (N,) ndarray
specifying offset for each slice of each frame, from the frame start time.

Returns

filist
[`FmriImageList` instance]

get_list_data(*axis=None*)

Return data in ndarray with list dimension at position *axis*

Parameters

axis
[int] *axis* specifies which axis of the output will take the role of the list dimension. For example, 0 will put the list dimension in the first axis of the result.

Returns

data
[ndarray] data in image list as array, with data across elements of the list concetenated at dimension *axis* of the array.

Examples

```
>>> from nipy.testing import funcfile
>>> from nipy.io.api import load_image
>>> funcim = load_image(funcfile)
>>> ilist = ImageList.from_image(funcim, axis='t')
>>> ilist.get_list_data(axis=0).shape
(20, 17, 21, 3)
```

nipy.modalities.fmri.fmri.**axis0_generator**(*data*, *slicers=None*)

Takes array-like *data*, returning slices over axes > 0

This function takes an array-like object *data* and yields tuples of slicing thing and slices like:

```
[slicer, np.asarray(data)[:,slicer] for slicer in slicer]
```

which in the default (*slicers* is None) case, boils down to:

```
[i, np.asarray(data)[:,i] for i in range(data.shape[1])]
```

This can be used to get arrays of time series out of an array if the time axis is axis 0.

Parameters

data
[array-like] object such that `arr = np.asarray(data)` returns an array of at least 2 dimensions.

slicers
[None or sequence] sequence of objects that can be used to slice into array `arr` returned from data. If None, default is `range(data.shape[1])`

CHAPTER

THREE

MODALITIES.FMRI.FMRISTAT.HRF

133.1 Module: `modalities.fmri.fmristat.hrf`

Computation of the canonical HRF used in fMRIstat, both the 2-term spectral approximation and the Taylor series approximation, to a shifted version of the canonical Glover HRF.

133.1.1 References

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.

133.2 Functions

`nipy.modalities.fmri.fmristat.hrf.`**`spectral_decomposition`**(*hrf2decompose*, *time=None*, *delta=None*, *ncomp=2*)

PCA decomposition of symbolic HRF shifted over time

Perform a PCA expansion of a symbolic HRF, time shifted over the values in delta, returning the first ncomp components.

This smooths out the HRF as compared to using a Taylor series approximation.

Parameters

hrf2decompose
[sympy expression] An expression that can be lambdified as a function of 't'. This is the HRF to be expanded in PCA

time
[None or np.ndarray, optional] None gives default value of np.linspace(-15,50,3251) chosen to match fMRIstat implementation. This corresponds to a time interval of 0.02. Presumed to be equally spaced.

delta
[None or np.ndarray, optional] None results in default value of np.arange(-4.5, 4.6, 0.1) chosen to match fMRIstat implementation.

ncomp
[int, optional] Number of principal components to retain.

Returns

hrf
[[sympy expressions]] A sequence length *ncomp* of symbolic HRFs that are the principal components.

approx
TODO

`nipy.modalities.fmri.fmristat.hrf.`**`taylor_approx`**`(`*hrf2decompose*, *time=None*, *delta=None*`)`

A Taylor series approximation of an HRF shifted by times *delta*

Returns original HRF and gradient of HRF

Parameters

hrf2decompose
[sympy expression] An expression that can be lambdified as a function of 't'. This is the HRF to be expanded in PCA

time
[None or np.ndarray, optional] None gives default value of np.linspace(-15,50,3251) chosen to match fMRIstat implementation. This corresponds to a time interval of 0.02. Presumed to be equally spaced.

delta
[None or np.ndarray, optional] None results in default value of np.arange(-4.5, 4.6, 0.1) chosen to match fMRIstat implementation.

Returns

hrf
[[sympy expressions]] Sequence length 2 comprising (*hrf2decompose*, `dhrf`) where `dhrf` is the first derivative of *hrf2decompose*.

approx
TODO

References

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.

CHAPTER

FOUR

MODALITIES.FMRI.FMRISTAT.INVERT

134.1 Module: `modalities.fmri.fmristat.invert`

`nipy.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.

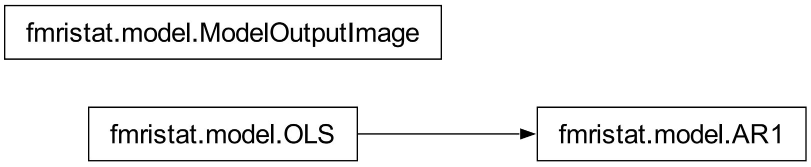
CHAPTER

FIVE

MODALITIES.FMRI.FMRISTAT.MODEL

135.1 Module: `modalities.fmri.fmristat.model`

Inheritance diagram for `nipy.modalities.fmri.fmristat.model`:



This module defines the two default GLM passes of fmristat

The results of both passes of the GLM get pushed around by generators, which know how to get out the (probably 3D) data for each slice, or parcel (for the AR) case, estimate in 2D, then store the data back again in its original shape.

The containers here, in the execute methods, know how to reshape the data on the way into the estimation (to 2D), then back again, to 3D, or 4D.

It's relatively easy to do this when just iterating over simple slices, but it gets a bit more complicated when taking arbitrary shaped samples from the image, as we do for estimating the AR coefficients, where we take all the voxels with similar AR coefficients at once.

135.2 Classes

135.2.1 `AR1`

class `nipy.modalities.fmri.fmristat.model.`**`AR1`**(*fmri_image*, *formula*, *rho*, *outputs=None*, *volume_start_times=None*)

Bases: *OLS*

Second pass through fmri_image.

Parameters

fmri_image
[*FmriImageList*] object returning 4D array from `np.asarray`, having attribute `volume_start_times` (if *volume_start_times* is None), and such that `object[0]` returns something with attributes `shape`

formula
[`nipy.algorithms.statistics.formula.Formula`]

rho
[`Image`] image of AR(1) coefficients. Returning data from `rho.get_fdata()`, and having attribute `coordmap`

outputs
[list] Store for model outputs.

volume_start_times: None or float or (N,) ndarray
start time of each frame. It can be specified either as an ndarray with `N=len(images)` elements or as a single float, the TR. None results in `np.arange(len(images)).astype(np.float64)`

Raises

ValueError
If *volume_start_times* not specified, and 4D image passed.

`__init__`(*fmri_image*, *formula*, *rho*, *outputs=None*, *volume_start_times=None*)

`execute`()

135.2.2 `ModelOutputImage`

`class nipy.modalities.fmri.fmristat.model.`**`ModelOutputImage`**(*filename*, *coordmap*, *shape*, *clobber=False*)

Bases: `object`

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

135.2.3 `OLS`

`class nipy.modalities.fmri.fmristat.model.`**`OLS`**(*fmri_image*, *formula*, *outputs=None*, *volume_start_times=None*)

Bases: `object`

First pass through fmri_image.

Parameters

fmri_image
[*FmriImageList* or 4D image] object returning 4D data from np.asarray, with first (`object[0]`) axis being the independent variable of the model; object[0] returns an object with attribute `shape`.

formula
[`nipy.algorithms.statistics.formula.Formula`]

outputs
[list] Store for model outputs.

volume_start_times: None or float or (N,) ndarray
start time of each frame. It can be specified either as an ndarray with `N=len(images)` elements or as a single float, the TR. None results in `np.arange(len(images)).astype(np.float64)`

Raises

ValueError
If *volume_start_times* not specified, and 4D image passed.

`__init__`(*fmri_image*, *formula*, *outputs=None*, *volume_start_times=None*)

`execute`()

135.3 Functions

`nipy.modalities.fmri.fmristat.model.`**`estimateAR`**(*resid*, *design*, *order=1*)
Estimate AR parameters using bias correction from fMRIstat.

Parameters

resid: array-like
residuals from model

model: an OLS model used to estimate residuals

Returns

output
[array] shape (order, resid

`nipy.modalities.fmri.fmristat.model.`**`generate_output`**(*outputs*, *iterable*, *reshape=<function <lambda>>*)
Write out results of a given output.

In the regression setting, results is generally going to be a scipy.stats.models.model.LikelihoodModelResults instance.

Parameters

outputs
[sequence] sequence of output objects

iterable
[object] Object which iterates, returning tuples of (indexer, results), where `indexer` can be used to index into the *outputs*

reshape
[callable] accepts two arguments, first is the indexer, and the second is the array which will be indexed; returns modified indexer and array ready for slicing with modified indexer.

`nipy.modalities.fmri.fmristat.model.`**`model_generator`**`(`*formula*, *data*, *volume_start_times*, *iterable=None*, *slicetimes=None*, *model_type=<class 'nipy.algorithms.statistics.models.regression.OLSModel'>*, *model_params=<function <lambda>>*`)`

Generator for the models for a pass of fmristat analysis.

`nipy.modalities.fmri.fmristat.model.`**`output_AR1`**`(`*outfile*, *fmri_image*, *clobber=False*`)`

Create an output file of the AR1 parameter from the OLS pass of fmristat.

Parameters

outfile
fmri_image
[`FmriImageList` or 4D image] object such that `object[0]` has attributes `coordmap` and `shape`

clobber
[bool] if True, overwrite previous output

Returns

regression_output
[`RegressionOutput` instance]

`nipy.modalities.fmri.fmristat.model.`**`output_F`**`(`*outfile*, *contrast*, *fmri_image*, *clobber=False*`)`

output F statistic images

Parameters

outfile
[str] filename for F contrast image

contrast
[array] F contrast matrix

fmri_image
[`FmriImageList` or `Image`] object such that `object[0]` has attributes `shape` and `coordmap`

clobber
[bool] if True, overwrites previous output; if False, raises error

Returns

f_reg_out
[`RegressionOutput` instance] Object that can a) be called with a results instance as argument, returning an array, and b) accept the output array for storing, via `obj[slice_spec] = arr` type slicing.

`nipy.modalities.fmri.fmristat.model.`**`output_T`**`(`*outbase*, *contrast*, *fmri_image*, *effect=True*, *sd=True*, *t=True*, *clobber=False*`)`

Return t contrast regression outputs list for *contrast*

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
[array] F contrast matrix

fmri_image
[`FmriImageList` or `Image`] object such that `object[0]` has attributes `shape` and `coordmap`

effect
[{True, False}, optional] whether to write an effect image

sd
[{True, False}, optional] whether to write a standard deviation image

t
[{True, False}, optional] whether to write a t image

clobber
[{False, True}, optional] whether to overwrite images that exist.

Returns

reglist
[`RegressionOutputList` instance] Regression output list with selected outputs, where selection is by inputs *effect*, *sd* and *t*

Notes

Note that this routine uses the corresponding `output_T` routine in `outputters`, but indirectly via the TOutput object.

`nipy.modalities.fmri.fmristat.model.`**`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.

Parameters

outfile
fmri_image
[`FmriImageList` or 4D image] If `FmriImageList`, needs attributes `volume_start_times`, supports len(), and object[0] has attributes `affine`, `coordmap` and `shape`, from which we create a new 4D coordmap and shape If 4D image, use the images coordmap and shape

clobber
[bool] if True, overwrite previous output

Returns

regression_output

`nipy.modalities.fmri.fmristat.model.`**`results_generator`**(*model_iterable*)

Generator for results from an iterator that returns (index, data, model) tuples.

See model_generator.

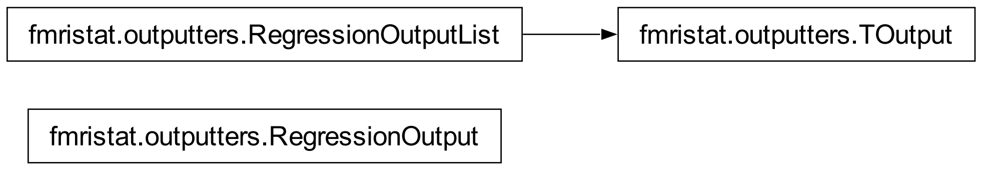
CHAPTER

SIX

MODALITIES.FMRI.FMRISTAT.OUTPUTTERS

136.1 Module: `modalities.fmri.fmristat.outputters`

Inheritance diagram for `nipy.modalities.fmri.fmristat.outputters`:


fmristat.outputters.RegressionOutputList
fmristat.outputters.TOutput
fmristat.outputters.RegressionOutput

Convenience functions and classes for statistics on images.

These functions and classes support the return of statistical test results from iterations through data.

The basic container here is the RegressionOutput. This does two basic things:

- via __call__, processes a result object from a regression to produce something, usually an array
- via slicing (__setitem__), it can store stuff, usually arrays.

We use these by other objects (see algorithms.statistics.fmri.fmristat) slicing data out of images, fitting models to the data to create results objects, and then passing them to these here `RegressionOutput` containers via call, to get useful arrays, and then putting the results back into the `RegressionOutput` containers via slicing (__setitem__).

136.2 Classes

136.2.1 `RegressionOutput`

class `nipy.modalities.fmri.fmristat.outputters.`**`RegressionOutput`**(*img*, *fn*, *output_shape=None*)

Bases: `object`

A class to output things in GLM passes through arrays of data.

`__init__`(*img*, *fn*, *output_shape=None*)

Parameters

img
[Image instance] The output Image

fn
[callable] A function that is applied to a models.model.LikelihoodModelResults instance

136.2.2 RegressionOutputList

class nipy.modalities.fmri.fmristat.outputters.**RegressionOutputList**(*imgs*, *fn*)

Bases: object

A class to output more than one thing from a GLM pass through arrays of data.

__init__(*imgs*, *fn*)

Initialize regression output list

Parameters

imgs
[list] The list of output images

fn
[callable] A function that is applied to a models.model.LikelihoodModelResults instance

136.2.3 TOutput

class nipy.modalities.fmri.fmristat.outputters.**TOutput**(*contrast*, *effect=None*, *sd=None*, *t=None*)

Bases: *RegressionOutputList*

Output contrast related to a T contrast from a GLM pass through data.

__init__(*contrast*, *effect=None*, *sd=None*, *t=None*)

Initialize regression output list

Parameters

imgs
[list] The list of output images

fn
[callable] A function that is applied to a models.model.LikelihoodModelResults instance

136.3 Functions

nipy.modalities.fmri.fmristat.outputters.**output_AR1**(*results*)

Compute the usual AR(1) parameter on the residuals from a regression.

nipy.modalities.fmri.fmristat.outputters.**output_F**(*results*, *contrast*)

This convenience function outputs the results of an Fcontrast from a regression

Parameters

results
[object] implementing Tcontrast method

contrast
[array] contrast matrix

Returns

F
[array] array of F values

`nipy.modalities.fmri.fmristat.outputters.`**`output_T`**(*results*, *contrast*, *retvals=('effect', 'sd', 't')*)

Convenience function to collect t contrast results

Parameters

results
[object] implementing Tcontrast method

contrast
[array] contrast matrix

retvals
[sequence, optional] None or more of strings 'effect', 'sd', 't', where the presence of the string means that that output will be returned.

Returns

res_list
[list] List of results. It will have the same length as *retvals* and the elements will be in the same order as retvals

`nipy.modalities.fmri.fmristat.outputters.`**`output_resid`**(*results*)

This convenience function outputs the residuals from a regression

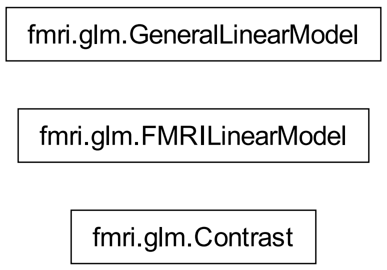
CHAPTER

SEVEN

MODALITIES.FMRI.GLM

137.1 Module: modalities.fmri.glm

Inheritance diagram for `nipy.modalities.fmri.glm`:



This module presents an interface to use the glm implemented in nipy.algorithms.statistics.models.regression.

It contains the GLM and contrast classes that are meant to be the main objects of fMRI data analyses.

It is important to note that the GLM is meant as a one-session General Linear Model. But inference can be performed on multiple sessions by computing fixed effects on contrasts

137.1.1 Examples

```
>>> import numpy as np
>>> from nipy.modalities.fmri.glm import GeneralLinearModel
>>> n, p, q = 100, 80, 10
>>> X, Y = np.random.randn(p, q), np.random.randn(p, n)
>>> cval = np.hstack((1, np.zeros(9)))
>>> model = GeneralLinearModel(X)
>>> model.fit(Y)
>>> z_vals = model.contrast(cval).z_score() # z-transformed statistics
```

Example of fixed effects statistics across two contrasts

```
>>> cval_ = cval.copy()
>>> np.random.shuffle(cval_)
>>> z_ffx = (model.contrast(cval) + model.contrast(cval_)).z_score()
```

137.2 Classes

137.2.1 Contrast

class nipy.modalities.fmri.glm.**Contrast**(*effect*, *variance*, *dof=10000000000.0*, *contrast_type='t'*, *tiny=1e-50*, *dofmax=10000000000.0*)

Bases: `object`

The contrast class handles the estimation of statistical contrasts on a given model: student (t), Fisher (F), conjunction (tmin-conjunction). The important feature is that it supports addition, thus opening the possibility of fixed-effects models.

The current implementation is meant to be simple, and could be enhanced in the future on the computational side (high-dimensional F contrasts may lead to memory breakage).

Notes

The ‘tmin-conjunction’ test is the valid conjunction test discussed in: Nichols T, Brett M, Andersson J, Wager T, Poline JB. Valid conjunction inference with the minimum statistic. Neuroimage. 2005 Apr 15;25(3):653-60. This test gives the p-value of the z-values under the conjunction null, i.e. the union of the null hypotheses for all terms.

__init__(*effect*, *variance*, *dof=10000000000.0*, *contrast_type='t'*, *tiny=1e-50*, *dofmax=10000000000.0*)

Parameters

effect: array of shape (contrast_dim, n_voxels)
the effects related to the contrast

variance: array of shape (contrast_dim, contrast_dim, n_voxels)
the associated variance estimate

dof: scalar, the degrees of freedom
contrast_type: string to be chosen among ‘t’ and ‘F’

p_value(*baseline=0.0*)

Return a parametric estimate of the p-value associated with the null hypothesis: (H0) ‘contrast equals baseline’

Parameters

baseline: float, optional
Baseline value for the test statistic

Notes

The value of 0.5 is used where the stat is not defined

`stat`(*baseline=0.0*)

Return the decision statistic associated with the test of the null hypothesis: (H0) 'contrast equals baseline'

Parameters

baseline: float, optional,
Baseline value for the test statistic

`z_score`(*baseline=0.0*)

Return a parametric estimation of the z-score associated with the null hypothesis: (H0) 'contrast equals baseline'

Parameters

baseline: float, optional
Baseline value for the test statistic

Notes

The value of 0 is used where the stat is not defined

137.2.2 `FMRILinearModel`

class `nipy.modalities.fmri.glm.`**`FMRILinearModel`**(*fmri_data*, *design_matrices*, *mask='compute'*, *m=0.2*, *M=0.9*, *threshold=0.5*)

Bases: `object`

This class is meant to handle GLMs from a higher-level perspective i.e. by taking images as input and output

`__init__`(*fmri_data*, *design_matrices*, *mask='compute'*, *m=0.2*, *M=0.9*, *threshold=0.5*)

Load the data

Parameters

fmri_data
[Image or str or sequence of Images / str] fmri images / paths of the (4D) fmri images

design_matrices
[arrays or str or sequence of arrays / str] design matrix arrays / paths of .npz files

mask
[str or Image or None, optional] string can be 'compute' or a path to an image image is an input (assumed binary) mask image(s), if 'compute', the mask is computed if None, no masking will be applied

m, M, threshold: float, optional
parameters of the masking procedure. Should be within [0, 1]

Notes

The only computation done here is mask computation (if required)

Examples

We need the example data package for this example:

```
from nipy.utils import example_data
from nipy.modalities.fmri.glm import FMRILinearModel
fmri_files = [example_data.get_filename('fiac', 'fiac0', run)
    for run in ['run1.nii.gz', 'run2.nii.gz']]
design_files = [example_data.get_filename('fiac', 'fiac0', run)
    for run in ['run1_design.npz', 'run2_design.npz']]
mask = example_data.get_filename('fiac', 'fiac0', 'mask.nii.gz')
multi_session_model = FMRILinearModel(fmri_files,
                                      design_files,
                                      mask)
multi_session_model.fit()
z_image, = multi_session_model.contrast([np.eye(13)[1]] * 2)

# The number of voxels with p < 0.001 given by ...
print(np.sum(z_image.get_fdata() > 3.09))
```

contrast(*contrasts*, *con_id=''*, *contrast_type=None*, *output_z=True*, *output_stat=False*, *output_effects=False*, *output_variance=False*)

Estimation of a contrast as fixed effects on all sessions

Parameters

contrasts
[array or list of arrays of shape (n_col) or (n_dim, n_col)] where `n_col` is the number of columns of the design matrix, numerical definition of the contrast (one array per run)

con_id
[str, optional] name of the contrast

contrast_type
[{'t', 'F', 'tmin-conjunction'}, optional] type of the contrast

output_z
[bool, optional] Return or not the corresponding z-stat image

output_stat
[bool, optional] Return or not the base (t/F) stat image

output_effects
[bool, optional] Return or not the corresponding effect image

output_variance
[bool, optional] Return or not the corresponding variance image

Returns

output_images
[list of nibabel images] The required output images, in the following order: z image, stat(t/F) image, effects image, variance image

`fit`(*do_scaling=True*, *model='ar1'*, *steps=100*)

Load the data, mask the data, scale the data, fit the GLM

Parameters

do_scaling
[bool, optional] if True, the data should be scaled as percent of voxel mean

model
[string, optional,] the kind of glm ('ols' or 'ar1') you want to fit to the data

steps
[int, optional] in case of an ar1, discretization of the ar1 parameter

137.2.3 GeneralLinearModel

class `nipy.modalities.fmri.glm.`**`GeneralLinearModel`**(*X*)

Bases: `object`

This class handles the so-called on General Linear Model

Most of what it does in the fit() and contrast() methods fit() performs the standard two-step ('ols' then 'ar1') GLM fitting contrast() returns a contrast instance, yileding statistics and p-values. The link between fit() and contrast is done vis the two class members:

glm_results
[dictionary of nipy.algorithms.statistics.models.] regression.RegressionResults instances, describing results of a GLM fit

labels
[array of shape(n_voxels),] labels that associate each voxel with a results key

`__init__`(*X*)

Parameters

X
[array of shape (n_time_points, n_regressors)] the design matrix

`contrast`(*con_val*, *contrast_type=None*)

Specify and estimate a linear contrast

Parameters

con_val
[numpy.ndarray of shape (p) or (q, p)] where q = number of contrast vectors and p = number of regressors

contrast_type
[{None, 't', 'F' or 'tmin-conjunction'}, optional] type of the contrast. If None, then defaults to 't' for 1D *con_val* and 'F' for 2D *con_val*

Returns

con: Contrast instance

`fit`(*Y*, *model='ols'*, *steps=100*)

GLM fitting of a dataset using 'ols' regression or the two-pass

Parameters

Y
[array of shape(n_time_points, n_samples)] the fMRI data

model
[{'ar1', 'ols'}, optional] the temporal variance model. Defaults to 'ols'

steps
[int, optional] Maximum number of discrete steps for the AR(1) coef histogram

get_beta(*column_index=None*)

Accessor for the best linear unbiased estimated of model parameters

Parameters

column_index: int or array-like of int or None, optional
The indexed of the columns to be returned. if None (default behaviour), the whole vector is returned

Returns

beta: array of shape (n_voxels, n_columns)
the beta

get_logL()

Accessor for the log-likelihood of the model

Returns

logL: array of shape (n_voxels,)
the sum of square error per voxel

get_mse()

Accessor for the mean squared error of the model

Returns

mse: array of shape (n_voxels)
the sum of square error per voxel

137.3 Function

nipy.modalities.fmri.glm.**data_scaling**(*Y*)

Scaling of the data to have percent of baseline change columnwise

Parameters

Y: array of shape(n_time_points, n_voxels)
the input data

Returns

Y: array of shape (n_time_points, n_voxels),
the data after mean-scaling, de-meaning and multiplication by 100

mean
[array of shape (n_voxels,)] the data mean

CHAPTER

EIGHT

MODALITIES.FMRI.HEMODYNAMIC_MODELS

138.1 Module: `modalities.fmri.hemodynamic_models`

This module is for canonical hrf specification. Here we provide for SPM, Glover hrfs and finite timpulse response (FIR) models. This module closely follows SPM implementation

Author: Bertrand Thirion, 2011–2013

138.2 Functions

`nipy.modalities.fmri.hemodynamic_models.`**`compute_regressor`**(*exp_condition*, *hrf_model*, *frametimes*, *con_id='cond'*, *oversampling=16*, *fir_delays=None*, *min_onset=-24*)

This is the main function to convolve regressors with hrf model

Parameters

exp_condition: descriptor of an experimental condition

hrf_model: string, the hrf model to be used. Can be chosen among:
‘spm’, ‘spm_time’, ‘spm_time_dispersion’, ‘canonical’, ‘canonical_derivative’, ‘fir’

frametimes: array of shape (n):the sought

con_id: string, optional identifier of the condition

oversampling: int, optional, oversampling factor to perform the convolution

fir_delays: array-like of int, onsets corresponding to the fir basis

min_onset: float, optional
minimal onset relative to frametimes[0] (in seconds) events that start before frametimes[0] + min_onset are not considered

Returns

creg: array of shape(n_scans, n_reg): computed regressors sampled
at frametimes

reg_names: list of strings, corresponding regressor names

Notes

The different hemodynamic models can be understood as follows: 'spm': this is the hrf model used in spm 'spm_time': this is the spm model plus its time derivative (2 regressors) 'spm_time_dispersion': idem, plus dispersion derivative (3 regressors) 'canonical': this one corresponds to the Glover hrf 'canonical_derivative': the Glover hrf + time derivative (2 regressors) 'fir': finite impulse response basis, a set of delayed dirac models

> with arbitrary length. This one currently assumes regularly spaced frametimes (i.e. fixed time of repetition).

It is expected that spm standard and Glover model would not yield large differences in most cases.

`nipy.modalities.fmri.hemodynamic_models.`**`glover_hrf`**(*tr*, *oversampling=16*, *time_length=32.0*, *onset=0.0*)

Implementation of the Glover hrf model

Parameters

tr: float, scan repeat time, in seconds
oversampling: int, temporal oversampling factor, optional
time_length: float, hrf kernel length, in seconds
onset: float, onset of the response

Returns

hrf: array of shape(length / tr * oversampling, float),
hrf sampling on the oversampled time grid

`nipy.modalities.fmri.hemodynamic_models.`**`glover_time_derivative`**(*tr*, *oversampling=16*, *time_length=32.0*, *onset=0.0*)

Implementation of the flover time derivative hrf (dhrf) model

Parameters

tr: float, scan repeat time, in seconds
oversampling: int, temporal oversampling factor, optional
time_length: float, hrf kernel length, in seconds
onset: float, onset of the response

Returns

dhrf: array of shape(length / tr, float),
dhrf sampling on the provided grid

`nipy.modalities.fmri.hemodynamic_models.`**`spm_dispersion_derivative`**(*tr*, *oversampling=16*, *time_length=32.0*, *onset=0.0*)

Implementation of the SPM dispersion derivative hrf model

Parameters

tr: float, scan repeat time, in seconds
oversampling: int, temporal oversampling factor, optional
time_length: float, hrf kernel length, in seconds
onset: float, onset of the response

Returns

dhrf: array of shape(length / tr * oversampling, float),
dhrf sampling on the oversampled time grid

`nipy.modalities.fmri.hemodynamic_models.`**`spm_hrf`**(*tr*, *oversampling=16*, *time_length=32.0*, *onset=0.0*)

Implementation of the SPM hrf model

Parameters

tr: float, scan repeat time, in seconds
oversampling: int, temporal oversampling factor, optional
time_length: float, hrf kernel length, in seconds
onset: float, onset of the response

Returns

hrf: array of shape(length / tr * oversampling, float),
hrf sampling on the oversampled time grid

`nipy.modalities.fmri.hemodynamic_models.`**`spm_time_derivative`**(*tr*, *oversampling=16*, *time_length=32.0*, *onset=0.0*)

Implementation of the SPM time derivative hrf (dhrf) model

Parameters

tr: float, scan repeat time, in seconds
oversampling: int, temporal oversampling factor, optional
time_length: float, hrf kernel length, in seconds
onset: float, onset of the response

Returns

dhrf: array of shape(length / tr, float),
dhrf sampling on the provided grid

CHAPTER

NINE

MODALITIES.FMRI.HRF

139.1 Module: `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.

The Glover HRF is based on:

@article{glover1999deconvolution,
  title={{Deconvolution of impulse response in event-related BOLD fMRI}}, author={Glover, G.H.}, journal={NeuroImage}, volume={9}, number={4}, pages={416–429}, year={1999}, publisher={Orlando, FL: Academic Press, c1992-}

}

This parametrization is from fmristat:

http://www.math.mcgill.ca/keith/fmristat/

fmristat models the HRF as the difference of two gamma functions, `g1` and `g2`, each defined by the timing of the gamma function peaks (`pk1, pk2`) and the FWHMs (`width1, width2`):

  raw_hrf = g1(pk1, width1) - a2 * g2(pk2, width2)

where `a2` is the scale factor for the `g2` gamma function. The actual hrf is the raw hrf set to have an integral of 1.

fmristat used `pk1, width1, pk2, width2, a2 = (5.4 5.2 10.8 7.35 0.35)`. These are parameters to match Glover's 1 second duration auditory stimulus curves. Glover wrote these as:

  y(t) = c1 * t**n1 * exp(t/t1) - a2 * c2 * t**n2 * exp(t/t2)

with `n1, t1, n2, t2, a2 = (6.0, 0.9, 12, 0.9, 0.35)`, and `c1, c2` being `1/max(t**n1 * exp(t/t1)), 1/max(t**n2 * exp(t/t2)`. The difference between Glover's expression and ours is because we (and fmristat) use the peak location and width to characterize the function rather than `n1, t1`. The values we use are equivalent. Specifically, in our formulation:

```
>>> n1, t1, c1 = gamma_params(5.4, 5.2)
>>> np.allclose((n1-1, t1), (6.0, 0.9), rtol=0.02)
True
>>> n2, t2, c2 = gamma_params(10.8, 7.35)
>>> np.allclose((n2-1, t2), (12.0, 0.9), rtol=0.02)
True
```

139.2 Functions

`nipy.modalities.fmri.hrf.`**`ddspmt`**(*t*)

SPM canonical HRF dispersion derivative, values for time values *t*

This is the canonical HRF dispersion derivative function as used in SPM.

It is the numerical difference between the HRF sampled at time *t*, and values at *t* for another HRF shape with a small change in the peak dispersion parameter (`peak_disp` in func:*spm_hrf_compat*).

`nipy.modalities.fmri.hrf.`**`dspmt`**(*t*)

SPM canonical HRF derivative, HRF derivative values for time values *t*

This is the canonical HRF derivative function as used in SPM.

It is the numerical difference of the HRF sampled at time *t* minus the values sampled at time *t* -1

`nipy.modalities.fmri.hrf.`**`gamma_expr`**(*peak_location*, *peak_fwhm*)

`nipy.modalities.fmri.hrf.`**`gamma_params`**(*peak_location*, *peak_fwhm*)

Parameters for gamma density given peak and width

TODO: where does the coef come from again.... check fmristat code

From a peak location and peak FWHM, determine the parameters (shape, scale) of a Gamma density:

f(x) = coef * x**(shape-1) * exp(-x/scale)

The coefficient returned ensures that the f has integral 1 over [0,np.inf]

Parameters

peak_location
[float] Location of the peak of the Gamma density

peak_fwhm
[float] FWHM at the peak

Returns

shape
[float] Shape parameter in the Gamma density

scale
[float] Scale parameter in the Gamma density

coef
[float] Coefficient needed to ensure the density has integral 1.

`nipy.modalities.fmri.hrf.`**`spm_hrf_compat`**(*t*, *peak_delay=6*, *under_delay=16*, *peak_disp=1*, *under_disp=1*, *p_u_ratio=6*, *normalize=True*)

SPM HRF function from sum of two gamma PDFs

This function is designed to be partially compatible with SPMs *spm_hrf.m* function.

The SPN HRF is a *peak* gamma PDF (with location *peak_delay* and dispersion *peak_disp*), minus an *undershoot* gamma PDF (with location *under_delay* and dispersion *under_disp*, and divided by the *p_u_ratio*).

Parameters

t
[array-like] vector of times at which to sample HRF.

peak_delay
[float, optional] delay of peak.

under_delay
[float, optional] delay of undershoot.

peak_disp
[float, optional] width (dispersion) of peak.

under_disp
[float, optional] width (dispersion) of undershoot.

p_u_ratio
[float, optional] peak to undershoot ratio. Undershoot divided by this value before subtracting from peak.

normalize
[{True, False}, optional] If True, divide HRF values by their sum before returning. SPM does this by default.

Returns

hrf
[array] vector length `len(t)` of samples from HRF at times *t*.

Notes

See `spm_hrf.m` in the SPM distribution.

`nipy.modalities.fmri.hrf.`**`spmt`**`(`*t*`)`

SPM canonical HRF, HRF values for time values *t*

This is the canonical HRF function as used in SPM

CHAPTER

MODALITIES.FMRI.REALFUNCS

140.1 Module: `modalities.fmri.realfuncs`

Helper functions for constructing design regressors

140.2 Functions

`nipy.modalities.fmri.realfuncs.`**`dct_ii_basis`**(*volume_times*, *order=None*, *normcols=False*)

DCT II basis up to order *order*

See: https://en.wikipedia.org/wiki/Discrete_cosine_transform#DCT-II

By default, basis not normalized to length 1, and therefore, basis is not orthogonal. Normalize basis with *normcols* keyword argument.

Parameters

volume_times
[array-like] Times of acquisition of each volume. Must be regular and continuous otherwise we raise an error.

order
[None or int, optional] Order of DCT-II basis. If None, return full basis set.

normcols
[bool, optional] If True, normalize columns to length 1, so return orthogonal *dct_basis*.

Returns

dct_basis
[array] Shape `(len(volume_times), order)` array with DCT-II basis up to order *order*.

Raises

ValueError
If difference between successive *volume_times* values is not constant over the 1D array.

`nipy.modalities.fmri.realfuncs.`**`dct_ii_cut_basis`**(*volume_times*, *cut_period*)

DCT-II regressors with periods >= *cut_period*

See: http://en.wikipedia.org/wiki/Discrete_cosine_transform#DCT-II

Parameters

volume_times
[array-like] Times of acquisition of each volume. Must be regular and continuous otherwise we raise an error.

cut_period: float
Cut period (wavelength) of the low-pass filter (in time units).

Returns

cdrift: array shape (n_scans, n_drifts)
DCT-II drifts plus a constant regressor in the final column. Constant regressor always present, regardless of *cut_period*.

CHAPTER

ONE

MODALITIES.FMRI.SPM.CORRELATION

141.1 Module: `modalities.fmri.spm.correlation`

141.2 Functions

`nipy.modalities.fmri.spm.correlation.`**`ARcomponents`**(*rho*, *n*, *drho=0.05*, *cor=False*, *sigma=1*)

Numerically differentiate covariance matrices of AR(p) of length n with respect to AR parameters around the value rho.

If drho is a vector, they are treated as steps in the numerical differentiation.

`nipy.modalities.fmri.spm.correlation.`**`ARcovariance`**(*rho*, *n*, *cor=False*, *sigma=1.0*)

Return covariance matrix of a sample of length n from an AR(p) process with parameters rho.

INPUTS:

rho – an array of length p sigma – standard deviation of the white noise

CHAPTER

TWO

MODALITIES.FMRI.SPM.MODEL

142.1 Module: `modalities.fmri.spm.model`

Inheritance diagram for `nipy.modalities.fmri.spm.model`:

142.2 Class

142.3 `SecondStage`

class `nipy.modalities.fmri.spm.model.`**`SecondStage`**(*fmri_image*, *formula*, *sigma*, *outputs=[]*, *volume_start_times=None*)

Bases: `object`

Parameters

fmri_image
[*FmriImageList*] object returning 4D array from `np.asarray`, having attribute `volume_start_times` (if *volume_start_times* is None), and such that `object[0]` returns something with attributes `shape`

formula
[`nipy.algorithms.statistics.formula.Formula`]

sigma
outputs
volume_start_times

`__init__`(*fmri_image*, *formula*, *sigma*, *outputs=[]*, *volume_start_times=None*)

`execute`()

142.4 Functions

`nipy.modalities.fmri.spm.model.`**`Fmask`**(*Fimg*, *dfnum*, *dfdenom*, *pvalue=0.0001*)

Create mask for use in estimating pooled covariance based on an F contrast.

`nipy.modalities.fmri.spm.model.`**`estimate_pooled_covariance`**(*resid*, *ARtarget=[0.3]*, *mask=None*)

Use SPM's REML implementation to estimate a pooled covariance matrix.

Thresholds an F statistic at a marginal pvalue to estimate covariance matrix.

CHAPTER

THREE

MODALITIES.FMRI.SPM.REML

143.1 Module: `modalities.fmri.spm.reml`

143.2 Functions

`nipy.modalities.fmri.spm.reml.`**`orth`**(*X*, *tol=1e-07*)

Compute orthonormal basis for the column span of X.

Rank is determined by zeroing all singular values, u, less than or equal to tol*u.max().

INPUTS:
X – n-by-p matrix

OUTPUTS:

B – n-by-rank(X) matrix with orthonormal columns spanning
the column rank of X

`nipy.modalities.fmri.spm.reml.`**`reml`**(*sigma*, *components*, *design=None*, *n=1*, *niter=128*, *penalty_cov=1.2664165549094176e-14*, *penalty_mean=0*)

Adapted from spm_reml.m

ReML estimation of covariance components from sigma using design matrix.

INPUTS:
sigma – m-by-m covariance matrix components – q-by-m-by-m array of variance components

mean of sigma is modeled as a some over components[i]

design – m-by-p design matrix whose effect is to be removed for
ReML. If None, no effect removed (???)

n – degrees of freedom of sigma penalty_cov – quadratic penalty to be applied in Fisher algorithm.

If the value is a float, f, the penalty is f * identity(m). If the value is a 1d array, this is the diagonal of the penalty.

penalty_mean – mean of quadratic penalty to be applied in Fisher
algorithm. If the value is a float, f, the location is f * np.ones(m).

OUTPUTS:
C – estimated mean of sigma h – array of length q representing coefficients

of variance components

cov_h – estimated covariance matrix of h

CHAPTER

FOUR

MODALITIES.FMRI.SPM.TRACE

144.1 Module: `modalities.fmri.spm.trace`

`nipy.modalities.fmri.spm.trace.`**`trRV`**(*X=None*, *V=None*)

If V is None it defaults to identity.

If X is None, it defaults to the 0-dimensional subspace, i.e. R is the identity.

```
>>> import numpy as np
>>> from numpy.random import standard_normal
>>>
>>> X = standard_normal((100, 4))
>>> np.allclose(trRV(X), (96.0, 96.0))
True
>>> V = np.identity(100)
>>> np.allclose(trRV(X), (96.0, 96.0))
True
>>>
>>> X[:,3] = X[:,1] + X[:,2]
>>> np.allclose(trRV(X), (97.0, 97.0))
True
>>>
>>> u = orth(X)
>>> V = np.dot(u, u.T)
>>> print(np.allclose(trRV(X, V), 0))
True
```

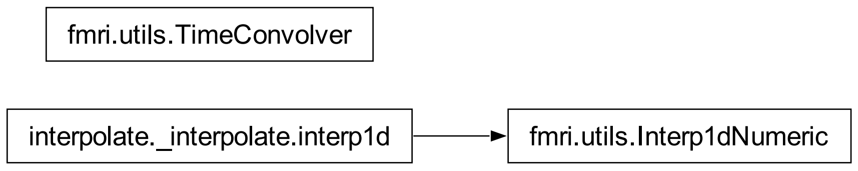
CHAPTER

FIVE

MODALITIES.FMRI.UTILS

145.1 Module: `modalities.fmri.utils`

Inheritance diagram for `nipy.modalities.fmri.utils`:



This module defines some convenience functions of time.

interp : an expression for a interpolated function of time

linear_interp
: [an expression for a linearly interpolated function of] time

step_function : an expression for a step function of time

events : a convenience function to generate sums of events

blocks : a convenience function to generate sums of blocks

convolve_functions : numerically convolve two functions of time

fourier_basis : a convenience function to generate a Fourier basis

145.2 Classes

145.2.1 `Interp1dNumeric`

class `nipy.modalities.fmri.utils.`**`Interp1dNumeric`**(*x*, *y*, *kind='linear'*, *axis=-1*, *copy=True*, *bounds_error=None*, *fill_value=nan*, *assume_sorted=False*)

Bases: `interp1d`

Wrapper for interp1 to raise TypeError for object array input

We need this because sympy will try to evaluate interpolated functions when constructing expressions involving floats. At least sympy 1.0 only accepts TypeError or AttributeError as indication that the implemented value cannot be sampled with the sympy expression. Therefore, raise a TypeError directly for an input giving an object array (such as a sympy expression), rather than letting interp1d raise a ValueError.

See:

- https://github.com/nipy/nipy/issues/395
- https://github.com/sympy/sympy/issues/10810

`__init__`(*x*, *y*, *kind='linear'*, *axis=-1*, *copy=True*, *bounds_error=None*, *fill_value=nan*, *assume_sorted=False*)

Initialize a 1-D linear interpolation class.

`dtype`

`property fill_value`

The fill value.

145.2.2 `TimeConvolver`

`class nipy.modalities.fmri.utils.`**`TimeConvolver`**(*expr*, *support*, *delta*, *fill=0*)

Bases: `object`

Make a convolution kernel from a symbolic function of t

A convolution kernel is a function with extra attributes to allow it to function as a kernel for numerical convolution (see *`convolve_functions()`*).

Parameters

expr
[sympy expression] An expression that is a function of t only.

support
[2 sequence] Sequence is `(low, high)` where expression is defined between `low` and `high`, and can be assumed to be *fill* otherwise

delta
[float] smallest change in domain of *expr* to use for numerical evaluation of *expr*

`__init__`(*expr*, *support*, *delta*, *fill=0*)

`convolve`(*g*, *g_interval*, *name=None*, ***kwargs*)

Convolve sympy expression *g* with this kernel

Parameters

g
[sympy expr] An expression that is a function of t only.

g_interval
[(2,) sequence of floats] Start and end of the interval of t over which to convolve g

name
[None or str, optional] Name of the convolved function in the resulting expression. Defaults to one created by `utils.interp`.

****kwargs**
[keyword args, optional] Any other arguments to pass to the `interp1d` function in creating the numerical function for *fg*.

Returns

fg
[sympy expr] An symbolic expression that is a function of t only, and that can be lambdified to produce a function returning the convolved series from an input array.

145.3 Functions

`nipy.modalities.fmri.utils.`**`blocks`**`(`*intervals*, *amplitudes=None*, *name=None*`)`

Step function based on a sequence of intervals.

Parameters

intervals
[(S,) sequence of (2,) sequences] Sequence (S0, S1, … S(N-1)) of sequences, where S0 (etc) are sequences of length 2, giving 'on' and 'off' times of block

amplitudes
[(S,) sequence of float, optional] Optional amplitudes for each block. Defaults to 1.

name
[None or str, optional] Name of the convolved function in the resulting expression. Defaults to one created by `utils.interp`.

Returns

b_of_t
[sympy expr] Sympy expression b(t) where b is a sympy anonymous function of time that implements the block step function

Examples

```
>>> on_off = [[1,2],[3,4]]
>>> tval = np.array([0.4,1.4,2.4,3.4])
>>> b = blocks(on_off)
>>> lam = lambdify_t(b)
>>> lam(tval)
array([ 0.,  1.,  0.,  1.])
>>> b = blocks(on_off, amplitudes=[3,5])
>>> lam = lambdify_t(b)
>>> lam(tval)
array([ 0.,  3.,  0.,  5.])
```

`nipy.modalities.fmri.utils.`**`convolve_functions`**`(`*f*, *g*, *f_interval*, *g_interval*, *dt*, *fill=0*, *name=None*, ***kwargs*`)`

Expression containing numerical convolution of *fn1* with *fn2*

Parameters

f
[sympy expr] An expression that is a function of t only.

g
: [sympy expr] An expression that is a function of t only.

f_interval
: [(2,) sequence of float] The start and end of the interval of t over which to convolve values of f

g_interval
: [(2,) sequence of floats] Start and end of the interval of t over which to convolve g

dt
: [float] Time step for discretization. We use this for creating the interpolator to form the numerical implementation

fill
: [None or float] Value to return from sampling output *fg* function outside range.

name
: [None or str, optional] Name of the convolved function in the resulting expression. Defaults to one created by `utils.interp`.

****kwargs**
: [keyword args, optional] Any other arguments to pass to the `interp1d` function in creating the numerical function for *fg*.

Returns

fg
: [sympy expr] An symbolic expression that is a function of t only, and that can be lambdified to produce a function returning the convolved series from an input array.

Examples

```
>>> from nipy.algorithms.statistics.formula.formulae import Term
>>> t = Term('t')
```

This is a square wave on (0,1)

```
>>> f1 = sympy.Piecewise((0, t <= 0), (1, t < 1), (0, True))
```

The convolution of `f1` with itself is a triangular wave on [0, 2], peaking at 1 with height 1

```
>>> tri = convolve_functions(f1, f1, [0, 2], [0, 2], 1.0e-3, name='conv')
```

The result is a symbolic function

```
>>> print(tri)
conv(t)
```

Get the numerical values for a time vector

```
>>> ftri = lambdify(t, tri)
>>> x = np.arange(0, 2, 0.2)
>>> y = ftri(x)
```

The peak is at 1 >>> x[np.argmax(y)] 1.0

`nipy.modalities.fmri.utils.`**`define`**`(`*name*, *expr*`)`

Create function of t expression from arbitrary expression *expr*

Take an arbitrarily complicated expression *expr* of 't' and make it an expression that is a simple function of t, of form `'%s(t)' % name` such that when it evaluates (via `lambdify`) it has the right values.

Parameters

expr
[sympy expression] with only 't' as a Symbol

name
[str]

Returns

nexpr: sympy expression

Examples

```
>>> t = Term('t')
>>> expr = t**2 + 3*t
>>> expr
t**2 + 3*t
>>> newexpr = define('f', expr)
>>> print(newexpr)
f(t)
>>> f = lambdify_t(newexpr)
>>> f(4)
28
>>> 3*4+4**2
28
```

`nipy.modalities.fmri.utils.`**`events`**`(`*times*, *amplitudes=None*, *f=DiracDelta*, *g=a*`)`

Return a sum of functions based on a sequence of times.

Parameters

times
[sequence] vector of onsets length N

amplitudes
[None or sequence length N, optional] Optional sequence of amplitudes. None (default) results in sequence length N of 1s

f
[sympy.Function, optional] Optional function. Defaults to DiracDelta, can be replaced with another function, f, in which case the result is the convolution with f.

g
[sympy.Basic, optional] Optional sympy expression function of amplitudes. The amplitudes, should be represented by the symbol 'a', which will be substituted, by the corresponding value in *amplitudes*.

Returns

sum_expression
[Sympy.Add] Sympy expression of time t, where onsets, as a function of t, have been symbolically convolved with function *f*, and any function g of corresponding amplitudes.

Examples

We import some sympy stuff so we can test if we've got what we expected

```
>>> from sympy import DiracDelta, Symbol, Function
>>> from nipy.modalities.fmri.utils import T
>>> evs = events([3,6,9])
>>> evs == DiracDelta(-9 + T) + DiracDelta(-6 + T) + DiracDelta(-3 + T)
True
>>> hrf = Function('hrf')
>>> evs = events([3,6,9], f=hrf)
>>> evs == hrf(-9 + T) + hrf(-6 + T) + hrf(-3 + T)
True
>>> evs = events([3,6,9], amplitudes=[2,1,-1])
>>> evs == -DiracDelta(-9 + T) + 2*DiracDelta(-3 + T) + DiracDelta(-6 + T)
True
```

nipy.modalities.fmri.utils.**fourier_basis**(*freq*)

sin and cos Formula for Fourier drift

The Fourier basis consists of sine and cosine waves of given frequencies.

Parameters

freq
[sequence of float] Frequencies for the terms in the Fourier basis.

Returns

f
[Formula]

Examples

```
>>> f=fourier_basis([1,2,3])
>>> f.terms
array([cos(2*pi*t), sin(2*pi*t), cos(4*pi*t), sin(4*pi*t), cos(6*pi*t),
       sin(6*pi*t)], dtype=object)
>>> f.mean
_b0*cos(2*pi*t) + _b1*sin(2*pi*t) + _b2*cos(4*pi*t) + _b3*sin(4*pi*t) + _
↪b4*cos(6*pi*t) + _b5*sin(6*pi*t)
```

nipy.modalities.fmri.utils.**interp**(*times*, *values*, *fill=0*, *name=None*, ***kw*)

Generic interpolation function of t given *times* and *values*

Imterpolator such that:

f(times[i]) = values[i]

if t < times[0] or t > times[-1]:
f(t) = fill

See `scipy.interpolate.interp1d` for details of interpolation types and other keyword arguments. Default is 'kind' is linear, making this function, by default, have the same behavior as `linear_interp`.

Parameters

times
[array-like] Increasing sequence of times

values
[array-like] Values at the specified times

fill
[None or float, optional] Value on the interval (-np.inf, times[0]). Default 0. If None, raises error outside bounds

name
[None or str, optional] Name of symbolic expression to use. If None, a default is used.

****kw**
[keyword args, optional] passed to `interp1d`

Returns

f
[sympy expression] A Function of t.

Examples

```
>>> s = interp([0,4,5.],[2.,4,6])
>>> tval = np.array([-0.1,0.1,3.9,4.1,5.1])
>>> res = lambdify_t(s)(tval)
```

0 outside bounds by default

```
>>> np.allclose(res, [0, 2.05, 3.95, 4.2, 0])
True
```

`nipy.modalities.fmri.utils.`**`lambdify_t`**`(`*expr*`)`

Return sympy function of t *expr* lambdified as function of t

Parameters

expr
[sympy expr]

Returns

func
[callable] Numerical implementation of function

`nipy.modalities.fmri.utils.`**`linear_interp`**`(`*times*, *values*, *fill=0*, *name=None*, ***kw*`)`

Linear interpolation function of t given *times* and *values*

Imterpolator such that:

f(times[i]) = values[i]

if t < times[0] or t > times[-1]:
f(t) = fill

This version of the function enforces the 'linear' kind of interpolation (argument to `scipy.interpolate.interp1d`).

Parameters

times
[array-like] Increasing sequence of times

values
[array-like] Values at the specified times

fill
[None or float, optional] Value on the interval (-np.inf, times[0]). Default 0. If None, raises error outside bounds

name
[None or str, optional] Name of symbolic expression to use. If None, a default is used.

****kw**
[keyword args, optional] passed to `interp1d`

Returns

f
[sympy expression] A Function of t.

Examples

```
>>> s = linear_interp([0,4,5.],[2.,4,6])
>>> tval = np.array([-0.1,0.1,3.9,4.1,5.1])
>>> res = lambdify_t(s)(tval)
```

0 outside bounds by default

```
>>> np.allclose(res, [0, 2.05, 3.95, 4.2, 0])
True
```

`nipy.modalities.fmri.utils.`**`step_function`**(*times*, *values*, *name=None*, *fill=0*)

Right-continuous step function of time t

Function of t such that

f(times[i]) = values[i]

if t < times[0]:
f(t) = fill

Parameters

times
[(N,) sequence] Increasing sequence of times

values
[(N,) sequence] Values at the specified times

fill
[float] Value on the interval (-np.inf, times[0])

name
[str] Name of symbolic expression to use. If None, a default is used.

Returns

f_t
[sympy expr] Sympy expression f(t) where f is a sympy implemented anonymous function

of time that implements the step function. To get the numerical version of the function, use `lambdify_t(f_t)`

Examples

```
>>> s = step_function([0,4,5],[2,4,6])
>>> tval = np.array([-0.1,3.9,4.1,5.1])
>>> lam = lambdify_t(s)
>>> lam(tval)
array([ 0.,  2.,  4.,  6.])
```

CHAPTER

SIX

PKG_INFO

146.1 Module: `pkg_info`

146.2 Functions

`nipy.pkg_info.`**`get_pkg_info`**`(`*pkg_path*`)`

Return dict describing the context of this package

Parameters

pkg_path
[str] path containing __init__.py for package

Returns

context
[dict] with named parameters of interest

`nipy.pkg_info.`**`pkg_commit_hash`**`(`*pkg_path*`)`

Get short form of commit hash given directory *pkg_path*

There should be a file called 'COMMIT_INFO.txt' in *pkg_path*. This is a file in INI file format, with at least one section: `commit hash`, and two variables `archive_subst_hash` and `install_hash`. The first has a substitution pattern in it which may have been filled by the execution of `git archive` if this is an archive generated that way. The second is filled in by the installation, if the installation is from a git archive.

We get the commit hash from (in order of preference):

- A substituted value in `archive_subst_hash`;
- A written commit hash value in `install_hash`;
- git's output, if we are in a git repository

If all these fail, we return a not-found placeholder tuple.

Parameters

pkg_path
[str] directory containing package

Returns

hash_from
[str] Where we got the hash from - description

hash_str
[str] short form of hash

CHAPTER

SEVEN

TESTING.DECORATORS

147.1 Module: `testing.decorators`

Extend numpy's decorators to use nipy's gui and data labels.

147.2 Functions

`nipy.testing.decorators.`**`if_datasource`**(*ds*, *msg*)

`nipy.testing.decorators.`**`if_example_data`**(*f*)

`nipy.testing.decorators.`**`if_templates`**(*f*)

`nipy.testing.decorators.`**`make_label_dec`**(*label*, *ds=None*)

Factory function to create a decorator that applies one or more labels.

Parameters

label

[str 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.

ds

[str] An optional docstring for the resulting decorator. If not given, a default docstring is auto-generated.

Returns

ldec

[function] A decorator.

Examples

```
>>> slow = make_label_dec('slow')
>>> print(slow.__doc__)
Labels a test as 'slow'
```

```
>>> rare = make_label_dec(['slow','hard'],
... "Mix labels 'slow' and 'hard' for rare tests")
>>> @rare
```

(continues on next page)

(continued from previous page)

```
... def f(): pass
...
>>>
>>> f.slow
True
>>> f.hard
True
```

`nipy.testing.decorators.`**`needs_mpl_agg`**(*func*)

Decorator requiring matplotlib with agg backend

`nipy.testing.decorators.`**`needs_review`**(*msg*)

Skip a test that needs further review.

Parameters

msg
[string] msg regarding the review that needs to be done

CHAPTER

EIGHT

UTILS

148.1 Module: `utils`

Inheritance diagram for `nipy.utils`:

General utilities for code support.

These are modules that we (broadly-speaking) wrote; packages that other people wrote, that we ship, go in the nipy.externals tree.

148.2 `VisibleDeprecationWarning`

class `nipy.utils.`**`VisibleDeprecationWarning`**

Bases: `UserWarning`

Visible deprecation warning.

Python does not show any DeprecationWarning by default. Sometimes we do want to show a deprecation warning, when the deprecation is urgent, or the usage is probably a bug.

`__init__`(**args*, ***kwargs*)

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

CHAPTER

NINE

UTILS.ARRAYS

149.1 Module: `utils.arrays`

Array utilities

`nipy.utils.arrays.`**`strides_from`**(*shape*, *dtype*, *order='C'*)

Return strides as for continuous array shape *shape* and given *dtype*

Parameters

shape
[sequence] shape of array to calculate strides from

dtype
[dtype-like] dtype specifier for array

order
[{'C', 'F'}, optional] whether array is C or FORTRAN ordered

Returns

strides
[tuple] sequence length `len(shape)` giving strides for continuous array with given *shape*, *dtype* and *order*

Examples

```
>>> strides_from((2,3,4), 'i4')
(48, 16, 4)
>>> strides_from((3,2), np.float64)
(16, 8)
>>> strides_from((5,4,3), np.bool_, order='F')
(1, 5, 20)
```

UTILS.PERLPIE

150.1 Module: `utils.perlpie`

Perform a global search and replace on the current directory *recursively*.

This a small python wrapper around the *perl -p -i -e* functionality. I **strongly recommend** running *perlpie* on files under source control. In this way it's easy to track your changes and if you discover your regular expression was wrong you can easily revert. I also recommend using *grin* to test your regular expressions before running *perlpie*.

150.1.1 Parameters

regex
[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.

150.1.2 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.

150.1.3 Examples

Replace all occurrences 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 occurrences of importing make_doctest_suite:

```
perlpie 'from\snipy\.utils\.testutils.*make_doctest_suite'
```

150.2 Functions

nipy.utils.perlpie.**check_deps**()

nipy.utils.perlpie.**main**()

nipy.utils.perlpie.**perl_dash_pie**(*oldstr*, *newstr*, *dry_run=None*)

Use perl to replace the oldstr with the newstr.

Examples

# To replace all occurrences of 'import numpy as N' with 'import numpy as np' from nipy.utils import perlpie perlpie.perl_dash_pie(r'imports+numpys+ass+N', 'import numpy as np') grind | xargs perl -pi -e 's/imports+numpys+ass+N/import numpy as np/g'

nipy.utils.perlpie.**print_extended_help**(*option*, *opt_str*, *value*, *parser*, **args*, ***kwargs*)

CHAPTER

ONE

UTILS.UTILITIES

151.1 Module: `utils.utilities`

Collection of utility functions and classes

Some of these come from the matplotlib `cbook` module with thanks.

151.2 Functions

`nipy.utils.utilities.`**`is_iterable`**`(`*obj*`)`

Return True if *obj* is iterable

`nipy.utils.utilities.`**`is_numlike`**`(`*obj*`)`

Return True if *obj* looks like a number

`nipy.utils.utilities.`**`seq_prod`**`(`*seq*, *initial=1*`)`

General product of sequence elements

Parameters

seq
[sequence] Iterable of values to multiply.

initial
[object, optional] Initial value

Returns

prod
[object] Result of ``initial * seq[0] * seq[1] .. ``.

Part VI

Publications

CHAPTER

TWO

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 THREE

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

CHAPTER

FOUR

SOFTWARE LICENSE

Except where otherwise noted, all NIPY software is licensed under a revised BSD license.

See our *Licensing* page for more details.

CHAPTER FIVE

DOCUMENTATION LICENSE

Except where otherwise noted, all NIPY documentation is licensed under a Creative Commons Attribution 3.0 License.

All code fragments in the documentation are licensed under our software license.

BIBLIOGRAPHY

[VTK4] Schroeder, Will, Ken Martin, and Bill Lorensen. (2006) *The Visualization Toolkit–An Object-Oriented Approach To 3D Graphics*. : Kitware, Inc.

[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.

[1] Gruntz, Dominik. A new algorithm for computing asymptotic series. In: Proc. 1993 Int. Symp. Symbolic and Algebraic Computation. 1993. pp. 239-244.

[2] Gruntz thesis - p90

[3] https://en.wikipedia.org/wiki/Asymptotic_expansion

[1] https://en.wikipedia.org/wiki/Algebraic_expression

[1] Gruntz, Dominik. A new algorithm for computing asymptotic series. In: Proc. 1993 Int. Symp. Symbolic and Algebraic Computation. 1993. pp. 239-244.

[2] Gruntz thesis - p90

[3] https://en.wikipedia.org/wiki/Asymptotic_expansion

[1] https://en.wikipedia.org/wiki/Algebraic_expression

[1] Gruntz, Dominik. A new algorithm for computing asymptotic series. In: Proc. 1993 Int. Symp. Symbolic and Algebraic Computation. 1993. pp. 239-244.

[2] Gruntz thesis - p90

[3] https://en.wikipedia.org/wiki/Asymptotic_expansion

[1] https://en.wikipedia.org/wiki/Algebraic_expression

[1] W. Green. “Econometric Analysis,” 5th ed., Pearson, 2003.

[1] W. Green. “Econometric Analysis,” 5th ed., Pearson, 2003.

[1] W. Green. “Econometric Analysis,” 5th ed., Pearson, 2003.

[1] W. Green. “Econometric Analysis,” 5th ed., Pearson, 2003.

[1] W. Green. “Econometric Analysis,” 5th ed., Pearson, 2003.

[Worsley2002] K.J. Worsley, C.H. Liao, J. Aston, V. Petre, G.H. Duncan, F. Morales, A.C. Evans (2002) A General Statistical Analysis for fMRI Data. Neuroimage 15:1:15

[Worsley2002] K.J. Worsley, C.H. Liao, J. Aston, V. Petre, G.H. Duncan, F. Morales, A.C. Evans (2002) A General Statistical Analysis for fMRI Data. Neuroimage 15:1:15

[Worsley1994] 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.

PYTHON MODULE INDEX

INDEX